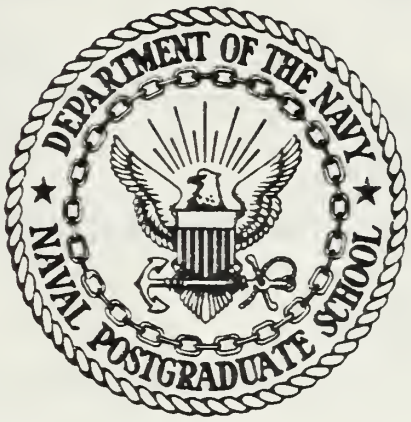
NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

INTEL 432/670 ADA BENCHMARK PERFORMANCE
EVALUATION IN THE
MULTIPROCESSOR/MULTIPROCESS ENVIRONMENT

by

Theodore F. Rogers, Jr.
and
Ioannis A. Karadimitropoulos
June, 1983

Thesis Advisor: Uno Kodres

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Intel 432/670 Ada Benchmark Performance Evaluation in the Multiprocessor/Multi-process Environment

5. TYPE OF REPORT & PERIOD COVERED

Master's Thesis
June, 1983

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Theodore F. Rogers, Jr.
Ioannis A. Karadimitropoulos

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Naval Postgraduate School
Monterey, California 93940

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Naval Postgraduate School
Monterey, California 93940

12. REPORT DATE

June, 1983

13. NUMBER OF PAGES

139

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

iAPX-432, INTEL, ADA, ADA-432
432/670 Cross Development System,
CFA, Computer Family Architecture,
Multiprocessor, and multiprocess

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The INTEL 432/670 microcomputer system contains the iAPX 432 microprocessor which executes compiled Ada programs. This thesis contains performance evaluation of the INTEL 432/670 system in a multiprocessor/multiprocess environment. Benchmark programs from the Computer Family Architecture (CFA) study are encoded in the Ada Programming Language and compiled on a host VAX 11/780 before being downloaded to INTEL MDS 800 (Cont)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

ABSTRACT (Continued) Block # 20

for further transfer to the INTEL 432/670 system for execution. The historical development of computer architectures as well as a systematic description of the INTEL 432/670 system are included.

Approved for public release; distribution unlimited

Intel 432/670 Ada Benchmark Performance Evaluation in the Multiprocessor/Multiprocess Environment

by

Theodore F. Rogers, Jr.
Captain, United States Navy
A.A., Potomac State College, 1960
B.S.E.S., Naval Postgraduate School, 1968

Ioannis A. Karadimitropoulos
Captain, Hellenic Army
B.S. Hellenic Military Academy, 1971
M.S.M.E. Engineering School of Athens, 1977

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1983

ABSTRACT

The INTEL 432/670 microcomputer system contains the iAPX 432 microprocessor which executes compiled Ada programs. This thesis contains performance evaluation of the INTEL 432/670 system in a multiprocessor/multiprocess environment. Benchmark programs from the Computer Family Architecture (CFA) study are encoded in the Ada Programming Language and compiled on a host VAX 11/780 before being downloaded to INTEL MDS 800 for further transfer to the INTEL 432/670 system for execution. The historical development of computer architectures as well as a systematic description of the INTEL 432/670 system are included.

TRADEMARKS

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempt to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis will be listed below, following the firm holding the trademark.

Intel corporation, Santa Clara, California:

Intel	MULTIBUS	INTELLEC MDS
Intel 8080	Intel 8086	iSBC 86/12A
Intel 4004	iAPX 432	Micromainframe

Digital Research, Pacific Grove, California:

CP/M

Motorola, INC:

EXORBUS

Department of Defense:

Ada

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

The authors would like to express an appreciation for the following personnel who have assisted with this project: Professor Uno R. Kodres, whose guidance, and analysis enabled the conclusion to be achieved through multiple paths. Professor Bruce J. Maclennan for making the concept of objects and their application to the Intel 432 understandable. Michael Williams whose superlative technical expertise enabled a timely completion of the project. Finally, to our wives and children for their support throughout the course of study.

I. INTRODUCTION

A. BACKGROUND

An Integrated Circuit (IC) is a small silicon semiconductor crystal containing electrical components such as transistors, diodes, resistors, capacitors, or interconnected digital gates. As the technology of IC's has improved, the number of components that can be put on a single silicon chip has increased. If a package contains several logic gates it is considered to be a small-scale integeration (SSI) device. Medium-scale integeration (MSI) devices contain 10 to 100 gates and perform a complete logic function. Large-scale integration (LSI) devices perform logic functions with over 100 devices and very-large-scale integration (VLSI) units contain thousands of gates in a single chip. The INTEL iAPX 432 contains 160,000 transistors on two chips and is considered a VLSI device.

B. THESIS DESCRIPTION

The Intel 432/670 system is a commercial product that uses the iAPX 432 microprocessor. Incorporating many new architectural features, with the goal to significantly reduce the software cost, the Intel 432 has been heralded as "the most significant architectural development in the last 5 - 10 years" [Ref. 1]. In addition to the VLSI technology, the 432 contains architectural advances that enable

implementation of transparent multiprocessing and a protected computing and communications environment.

The purpose of this thesis is to continue a series of benchmark performance tests on the Intel 432 that have been reported in Shoop [Ref. 2] and Applegate [Ref 3]. The goal of these tests is to determine if the 432/670 system is capable of sustaining incremental performance improvements as processors are added to the system.

C. THESIS ORGANIZATON

This thesis is composed of three major sections. Chapter II presents a brief history of computer architecture development from the first electronic computers to the object oriented architectures. Chapter III provides a systematic description of the Intel 432/670 system while Chapter IV provides the actual Benchmark performance evaluations of the Intel 432/670 system in the multiprocessor/multiprocess environment.

II. HISTORICAL DEVELOPMENT

Since computer systems architecture is not restricted to the sole aspect of hardware, it is not easily defined. Functional boxes and their complexity, the interconnection between the boxes, switches, controllers, the way messages are passed and received, and the blend of hardware and software features must be considered in the overall architectural structure. The Intel 432 architecture is the result of an evolutionary development in computer systems. It incorporates many advanced features which have evolved throughout the history of computers. Accordingly, to establish the proper perspective of the concepts of the Intel 432 a brief history of computer architecture development is presented.

One of the earlier electronic computers was completed in 1946 at the University of Pennsylvania Moore School of Engineering. ENIAC, capable of performing nearly 5000 additions or subtractions per second, consisted of more than 18000 vacuum tubes and 1500 relays and was used to compute ballistic trajectories. Wired for specific computations, any modifications or program replacements on ENIAC required new wiring and its associated excessive set up time. In an effort to eliminate the setup time, von Neumann and others, consulting at the Moore School of Engineering, designed the

first stored program computer. However, the first operational stored program computer (EDSAC) was built at Cambridge, England in 1949. In addition to being the first computer to execute stored programs, EDSAC introduced the notion of memory hierarchy through the primary memory and backing drum.

Meanwhile, von Neumann continued his research at the Institute for Advanced Studies (IAS) at Princeton where he developed the von Neumann architecture which is in wide use today. The von Neumann architecture is considered to be based on four key concepts:

1. A single, sequentially addressed memory. A program and its data are stored in a single memory, and the memory is referenced with sequential addresses.

2. A linear memory. The memory is one dimensional, or has the appearance of vector of words.

3. No explicit distinctions between instructions and data. Instructions and data are distinquished implicitly by the operations directed toward them.

4. Meaning is not an inherent part of data. [Ref. 4]

The early programming styles supported single user, non-subroutine environments that employed a looping technique by modifying instructions. Since instruction modification generated only serially reuseable code, multiprogramming was not feasible and index registers were developed at the

University of Manchester to make it possible to call subroutines, pass parameters, and generate reentrant code.

UNIVAC I, built for the Bureau of Census, had a tape system that used error checking procedures and buffering capabilities to read tapes both forward and backward. It was intended for both scientific and commercial applications and can be considered the first successful commercial computer. The system architecture during this period is characterized in Figure 2.1.

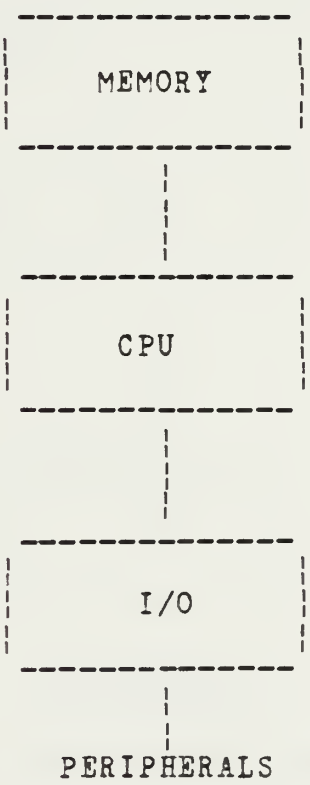


Figure 2.1 Early System Architecture

As computer applications advanced into commercial fields, I/O intensive requirements were generated.

Architecture improvements to enhance computational speeds and enable overlap to permit interleaved I/O and computational operations were developed. Pipelining, a technique used to speed computer operations by multiplexing memory accesses .pa with instruction decoding and execution, is one example of enhancement.

Systems became faster, smaller, and more reliable throughout the 1960's as transistors and integrated circuits allowed the building of more general purpose registers and regular architectures. Figures 2.2, 2.3, and 2.4 depict the architectures of this period.

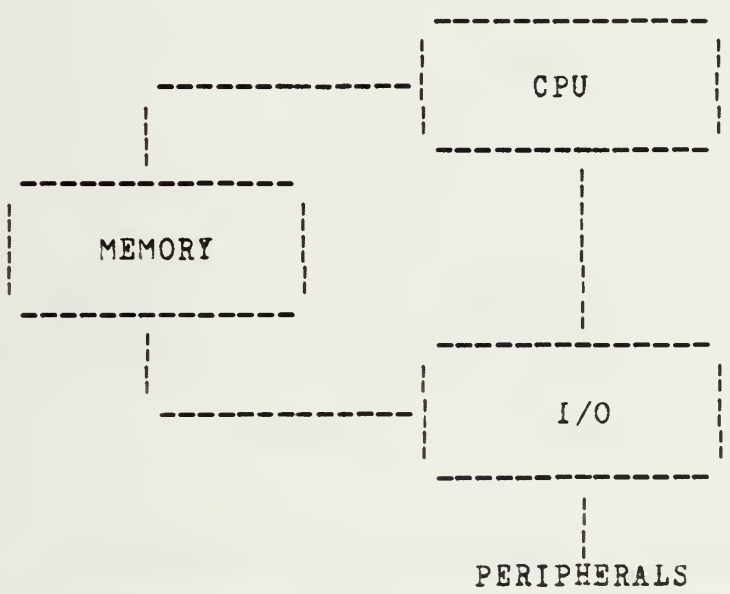


Figure 2.2 Dual Port Memory Architecture

Figure 2.2, representing a dual port memory architecture, provides a faster access path but demands extra hardware in the memory which performs independent I/O

operations. The I/O bus structure, Figure 2.3, connects the I/O devices to memory through device controllers that are, in many cases, mini-computers. It still requires a resource manager or a distributed control discipline. The central system controller shown in Figure 2.4 is based around a giant switch which controls multiple processors and memory units in a modular fashion.

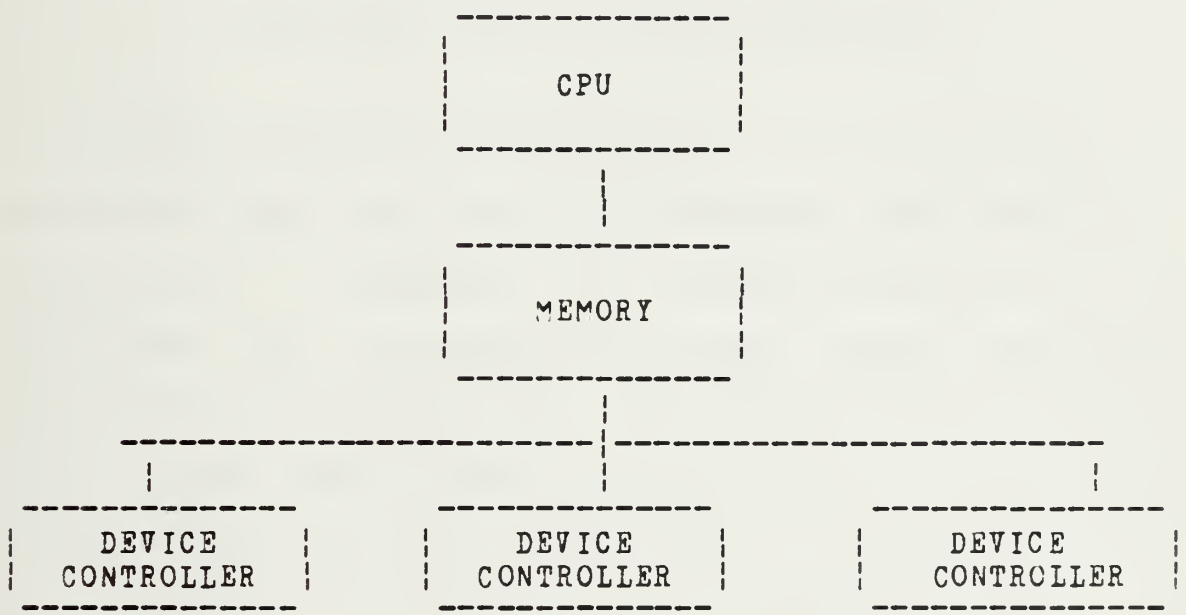


Figure 2.3 I/O Bus Architecture

As hardware cost decreased, designers eliminated the complicated central I/O control in favor of a bus oriented structure (Figure 2.5). It is flexible, modular, and conceptually simple and most architectures today contain variations of this bus oriented structure. S-100, MULTIBUS, EXORBUS, and VMEbus are a few bus oriented architectures on the current market.

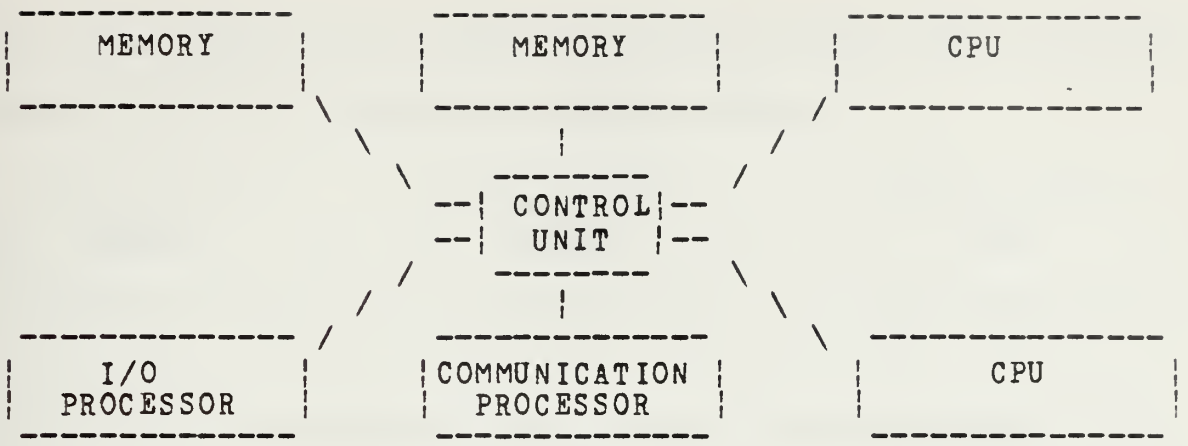


Figure 2.4 Central System Controller

As fast processors were developed, efforts to keep the processors busy, while the slow peripherals were reacting to instructions, culminated in multiprogramming and the associated task switching and hardware memory protection instruction set enhancements. Microprogramming, a technique which breaks each instruction into a series of elementary steps was also developed in the 1960's to reduce the cost of the sophisticated instruction sets by making it easier to develop a compatible series of computers that covered a wide range of performance.

Paging and segmentation were developed to support the concept of virtual memory. Additionally cache memory, which is fast memory physically close to the central processor, was developed to improve CPU performance. In general, the advances in memory management resulted in a five level hierarchy that consisted of registers, cache, main memory,

and high-speed disk or drum for backing store and low-speed disk or tape for long-term program storage.

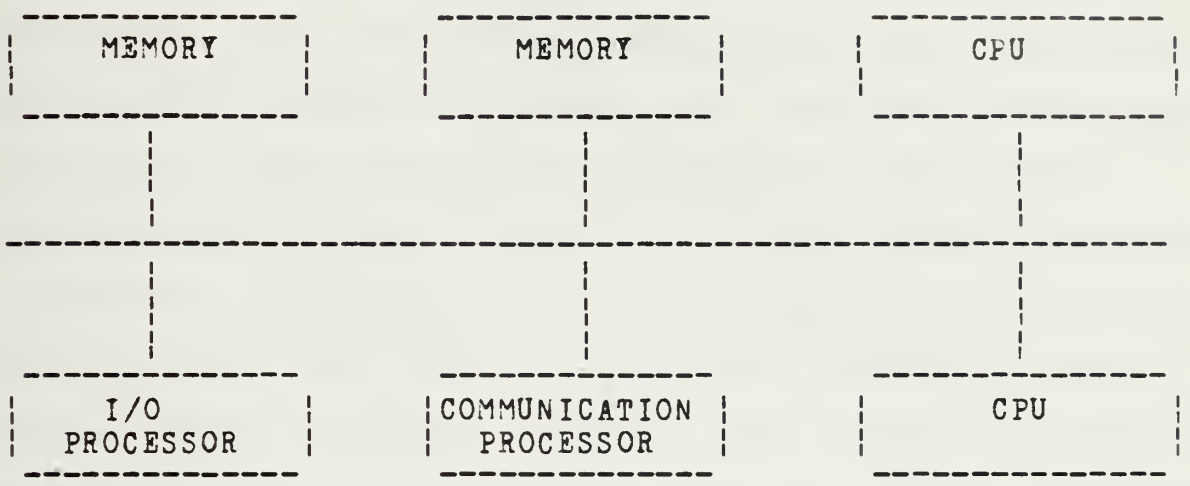


Figure 2.5 Single Bus Architecure

Multiprocessors, another concept introduced in the 1960's, are characterized architecturally by whether the processors are identical, execute the same instruction stream, or operate on one or more data streams. However, it was the Intel 4004 microprocessor which was developed in 1971 that was responsible for a continued unpresidented evolution in technology during the 1970's which produced the inexpensive processors and encouraged the use of multiprocesssing. Specialized database processors and arithmetic processors are a few examples of expansion in this area.

As complexity in microcomputer systems increased with decreasing hardware cost, software systems were expanding to take advantage of the new capabilities. Software engineers

soon discovered that large software systems were among the most difficult engineering projects. The use of subroutines, block structured languages, procedures, functions, stepwise refinement and structured programming were techniques developed to assist in solving the software engineering problems. Data abstraction, employing the concepts of encapsulation of data structures, uniform procedure/message interface, and decomposition of design, was a further extension of the effort to produce reliable software. While these software concepts were being developed, evolution in hardware protection structures can be observed in "supervisor vs. user state in the IBM 360 systems, hierarchical rings in Multics, and multiple domains in the Plessy s/250, CAL/TSS, Hydra, and CAP. Key concepts of these domain based systems include independent address spaces, capability-based addressing and access control, and the decomposition of the operating system into a useful set of abstractions." [Ref. 5] Additionally descriptors and access control concepts were introduced to provide protection by establishing a requirement for authorization to access data elements.

A major departure from one of the key concepts of the von Neumann architecture is observed in tagged storage. All information within storage is self-identifying. Based on the information contained in the tag field, the machine instructions can determine the semantics of each operation

as well as the data conversion rules to employ. With the tag field you have a strongly typed environment that prohibits operation on improper data types and permits the deferment of binding instructions to data attributes until the time of execution.

As we examined the history of computer development, we saw that the early machines used simple hardware operations such as "move byte" and "add integer" which manipulated hardward-recognized data types such as bytes and integers. As technology progressed and computer hardware gained functionality, more complicated operations such as floating point were moved into hardware, which substantially increases the speed of these operations. Enhancement of standard programming methods as well as guaranteed enforcement of data protection were additional benefits derived from moving software functions into hardware.

Recognizing that continued advancements in architectural enhancement would result in more data structures being moved into hardware, designers developed a methodology for viewing these abstractions that would contribute to the concept of raising the level of hardware/software interface. One such methodology that integrates data abstraction, domain based protection, and high level system functionality is the object oriented methodology.

The concept of using objects can be observed, as long ago as 1960, in LISP, where atoms have properties that can

be related to real world objects. SIMULA, an extension of ALGOL intended for simulation, included the idea of internally represented objects as instances of classes. In an effort to produce a programming vehicle that would reduce the man-machine semantic gap, Alan Kay, while still a graduate student at the University of Utah, investigated simulation and graphics oriented programming languages. Taking many of his ideas, such as classes and objects, from SIMULA, Kay helped design the language FLEX. Continued refinement of FLEX carried the object oriented paradigm to a smoother model by incorporating ideas from LOGO, a language designed to teach programming concepts to children, and resulted in the Smalltalk programming system.

An object is an abstraction that is usually considered to have the following characteristics:

1. objects are temporal; they exist in time;
2. objects are mutable (subject to change), and have a state;
3. objects can be created and destroyed;
4. objects are particular (distinct), and can be shared. [Ref. 6]

In general objects are similar to packages in that an object is a data structure that contains information in an organized manner, including a set of basic operations that directly manipulate the data structure. It can be referenced as one entity and has a label that tells its type.

As Kay continued his concept development throughout the 1970's, the idea of using an object oriented methodology to model complex real world issues began to emerge. IBM uses object oriented design concepts for complex projects. Grady Booch, formally of the USAF Academy, advocates an object oriented design using Ada to enforce a clear design structure. Just as Smalltalk had used objects to represent a desired level of abstraction, object-based architecture was selected to model the growing hardware complexities.

Based on an integrated approach to data abstraction and domain-based protection, object oriented architecture includes a form of capability based addressing and supports high level system functions such as memory management, process management, and processor management. Because the architecture is rooted in data abstraction, good software design methodology is encouraged which will result in software systems that will be more reliable and have reduced software life cycle cost. Elements of object-oriented architecture include:

1. Objects for system management and control.
2. Objects for program environments, flow of control, and intercommunication.
3. Facilities for user object extensions and system object type management.
4. Object addressing and protection mechanisms.
5. An instruction interface. [Ref. 7]

System management objects include processor objects, process objects and resource objects. While a processor object is a memory based data structure, process objects represent a unit of computational work for the processor and contain information regarding scheduling and dispatching. Various resource objects are queue-like structures used to bind physical processors with ready-to-run processes. By representing all resources as objects, a uniform interface for all system functions is obtained.

Environment and control objects include data objects, instruction objects, domain objects, context objects and control link objects, while communication objects consist of message and port objects. Port objects are asynchronous communication pathways between processes and provide for the queueing of waiting messages as well as resolution of speed disparity between concurrent activities in the system. An object oriented architecture with a built in software methodology should include support for representing typing, type manager packaging, and object authentication and creation. Object addressing and protection mechanisms are required to support references, mapping and protection. Finally, the object oriented instruction interface provides a uniform interface with uniform referencing to all the object based facilities.

A system environment can be viewed as a single set of objects which are addressable by everyone. Ranging from the

primitive storage segment having all the characteristics of a von Neumann store, to an abstract entity whose physical form is hidden by the architecture and whose meaning is defined solely by the transformations that can be performed on it, objects provide still more protection by encapsulating data and program modules in a structure with not only well defined external visibility but carefully hidden internal structure.

In summary, object oriented architectures were adapted to model the wide variety of hardware and software constructs that have been developed to enhance computer systems efficiency and reliability. In addition to raising the level at which the transition from hardware to software occurs, object oriented architecture enjoys technology independent design through its information hiding structures and supports a variety of language structures including operating systems and reliable low life cycle cost applications software. For these reasons, Intel Corporation has chosen the object-oriented architecture as the predominant construct of the iAPX 432.

III. INTEL 432/670 SYSTEM DESCRIPTION AND COMPONENT ARCHITECTURE

This chapter will present the characteristics and advantages of the iAPX432 over the conventional microcomputers. Additionally, the concepts of object oriented architecture will be presented in a simplified form.

A. THE 432 OBJECT ORIENTED ARCHITECTURE

In an effort to find a solution to the software crisis while providing transparent multiprocessing, the Intel 432 designers' principal desire was to provide direct execution time support for both data abstraction and domain-based operating systems. Recognizing that both of the objectives could be supported by the object model, Intel selected the object oriented architecture as the principal design construct to support a software methodology. This architecture provides mechanisms to help the programmer follow the object oriented methodology.

1. Object Oriented Methodology

What does object oriented methodology mean? One of the best ways to modularize programs is by using the principle of information hiding as defined by D. L. Parnas. The basic idea is that a module should contain a collection of related procedures and associated data structures on

which they operate. Procedures outside the module should not have access to the implementation details inside the module. The data can be referenced from outside the module only by a call to one of the procedures inside the module.

Modules that result when this principle is applied are referred to as abstract data types, extended types, or Parnas modules. Intel refers to them as type managers.

At the same time that programming languages research was focusing on the concept of a type manager as the solution to the modularization problem, operating system research was defining a very similar structure, the protection domain, as the solution to the security problem. Since operating system personnel were concerned more with security than modularization, they concentrated on the data in these domains, rather than the procedures which were the focus of the language research. Assigning the name "objects" to the data items associated with domains, operating systems personnel called the whole information hiding methodology object either the oriented methodology or the object model.

Although experimental studies have demonstrated that objects are superb models for modern software systems, the overhead required to maintain the model is too excessive to permit efficient implementation without direct hardware support. [Ref. 8.] The Intel 432 has hardware mechanisms that permit recognition of the following objects:

1. Processor
2. Processes
3. Dispatching ports
4. Interprocess messages
5. Domains
6. Context
7. Instruction
8. Data

to permit an effective management of the system's configuration and its individual resources as well as support for data abstraction.

B. INTEL 432/670 SYSTEM

Utilizing the advanced architecture of the iAPX 432 Micromainframe VLSI computer, the 432 family presents a new architecture in the world of microcomputers which contains the following salient characteristics.

1. Part of the operating system is contained in silicon and it is called The Silicon OS. It supports process scheduling, interprocess communication and dynamic storage allocation.

2. The system operates in both single processor and multiprocessor configurations and it can include up to eight (8) processors, although the 432/670 system can only support five (5) processors as limited by the number of slots available in the mother board. If one processor fails, the rest of the system can usually continue to operate.

3. Hardware recognition of eight (8) data types.

4. Hardware control functions for 16 and 32 bit multiply, divide, and remainder operations.

5. Hardwired control functions for 32, 64, and 80 bit floating point arithmetic.

6. Hardwired error correction and code protection.

7. Object oriented design.

8. Both abstract data types and objects are supported by hardware recognized, hardware protected and hardware manipulated structures.

9. It has an extremely large virtual address space (2**40 bytes) and hardware supplied mechanisms for implementing virtual memory systems that can exploit this address space.

The 432/670 system consists of the three (3) subsystems:

1. Memory subsystem. \ Central System
2. Processor subsystem. /
3. Peripheral subsystem.

and two busses:

1. System bus backplane.
2. Multibus backplane.

Before analyzing each one the subsystems, the following observations of the systems architecture are germane:

1. The whole memory is accessible by all GDPs.

2. The system is theoretically expandable to include any number of processors (GDPs).

3. 8 bit or 16 bit based systems can be connected to the system for performing I/O.

Figure 3.1 Illustrates the boundary between the central system and peripheral subsystems. In addition to seperating data processing from input/output processing, this boundary serves as a protection barrier. All information in central system memory is shielded by 432 hardware against unauthorized accesses but the peripheral subsystem may or may not provide protection.

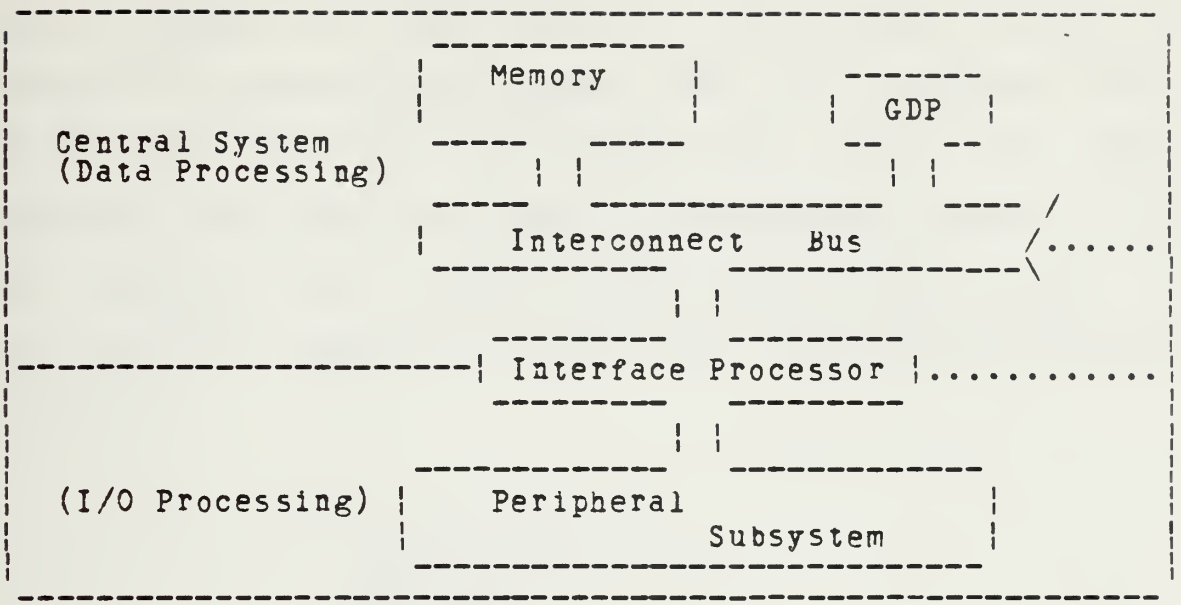


Figure 3.1 General 432/670 System Organization.

Figure 3.2 Illustrates a hypothetical configuration that employs two peripheral subsystems. The main system's hardware is composed of two GDPs and a common memory that is shared by these processors. The main system's software is a collection of one or more processes that execute on the GDP(s).

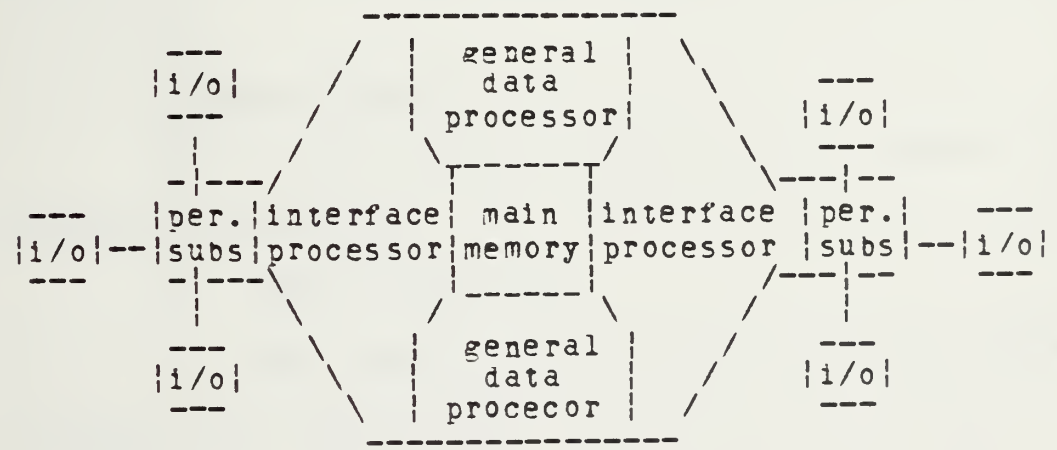


Figure 3.2 Main System and Peripheral Subsystem.

When the salient characteristics of the 432 are viewed with respect to the levels of architecture within a computing system cited by Meyers [Ref. 9], it becomes clear that functions have been moved from both high level language and operating system to the hardware which raise the boundary line between hardware and software closer to the end user. Figure 3.3 illustrates the hardware-software interface.

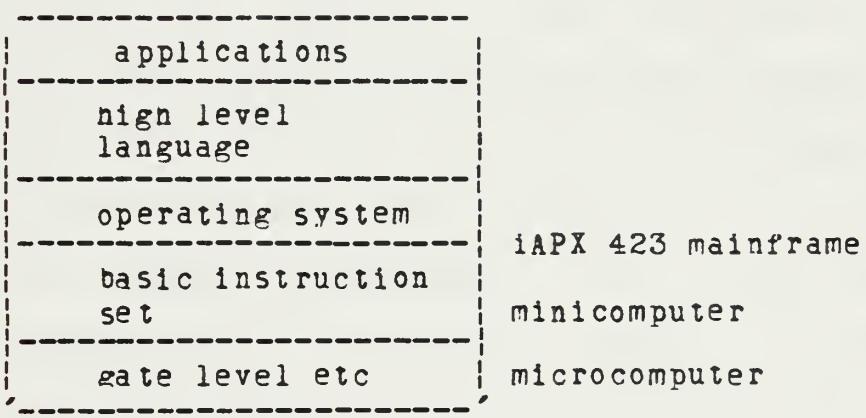


Figure 3.3 Hardware - Software Interface

Appendix A gives a comparison of the iAPX 432 architecture and the architecture of the conventional mainframes.

1. Central System

The central system consists of the memory and the processor subsystems.

a. Memory Subsystem

The memory subsystem resides on the system bus backplane and consists of:

1. One memory controller (MC) board.
2. One to Ten Storage Array (SA) boards.
3. A Memory Interleave (option).

(1) Memory Controller Board. The memory addressed by the 432 is organized in eight-bit bytes. In spite of this Intel says that the 432 is a 32 bit microprocessor and that it can perform 32, 64, and 80 bit floating point arithmetic because with any read request from the system processor(s), an additional signal determines how many bytes should be addressed. This is called byte addressability. Thus, based on the processor(s) requests a word can consist of 1, 2, 4, 8, or 10 bytes and consequently 8, 16, 32, 48, 64, or 80 bits can be addressed.

(2) Storage Array Board. Each Storage Array (SA) board contains 256K 8 bit bytes of memory. A memory controller can control up to 16 SA's or 4.096 Megabytes of memory. However, in the current 432/670 systems.there are only 10 slots available for SA boards.

Although the 432 is a 32 bit microprocessor the words are organized in 39 bit wide banks. The seven (7) additional bits contain Error Correction Code (ECD) for each 32 bit word. This is controlled by logic existing on every SA board for generating, checking and storing this ECD. Finally, the SA boards are organized in 64K byte banks of 32 bit words, four bytes per word.

(3) Interleave Option. This option can be selected by setting the jumpers of the MC board to the

proper position. The interleave option is available to enhance system performance. The interleave concept uses two banks of memory with alternating addresses to provide faster data access by overlapping memory operations. If the interleave option has been selected, SA boards must be installed in pairs to provide the alternate banks.

b. Processor Subsystem

The processor subsytem, which resides on the system bus backplane and consists of the GDP and IP boards, is discussed under component architecture.

c. Peripheral Subsystem

The peripheral subsystems are discussed in the Basic I/O Model and Windows sections.

C. COMPONENENT ARCHITECTURE

In this section, the basic architecture, of the major components of the system (GDP and IP) are presented.

1. Architecture

The 432 GDP consists of two chips the instruction decoder/microinstruction sequencer-43201 and the execution unit-43202. Appendix B contains the pin description of each chip while Figures 3.4, and 3.5 show the block diagrams of the 43201 and 43202 chips respectively.

In general, the GDP is organized internally as a three-stage microprogram-controlled pipeline. The first stage is the instruction decoder (ID), the microinstruction

sequencer (MS) is contained in the second stage while the third stage houses the execution unit.

The first two stages of the pipeline are physically located on the 43201 chip (Figure 3.4), while the third stage is the only function of the 43202 chip (Figure 3.5). Actually each stage of the pipeline is an independent subprocessor, which operates until the pipeline is full and then halts and waits for additional work.

a. Instruction Decoder

The first subprocessor of the pipeline is the Instuction Decoder (ID), which performs the following functions:

1. Receives macroinstructions.
2. Processes variable length ields.
3. Extracts logical addresses.
4. Generates starting addresses for micro-instruction procedures
5. Generates micro-instructions for simple operations.

The general task facing the Instruction Decoder is to interpret the macro-instruction stream, to determine which micro-instruction sequence should be initiated next and to extract logical address data. Additionally, the Instruction Decoder provides a mechanism to recover from an instruction that faults. The information necessary for fault recovery is retained by the Instruction Decoder until the instruction is successfully completed.

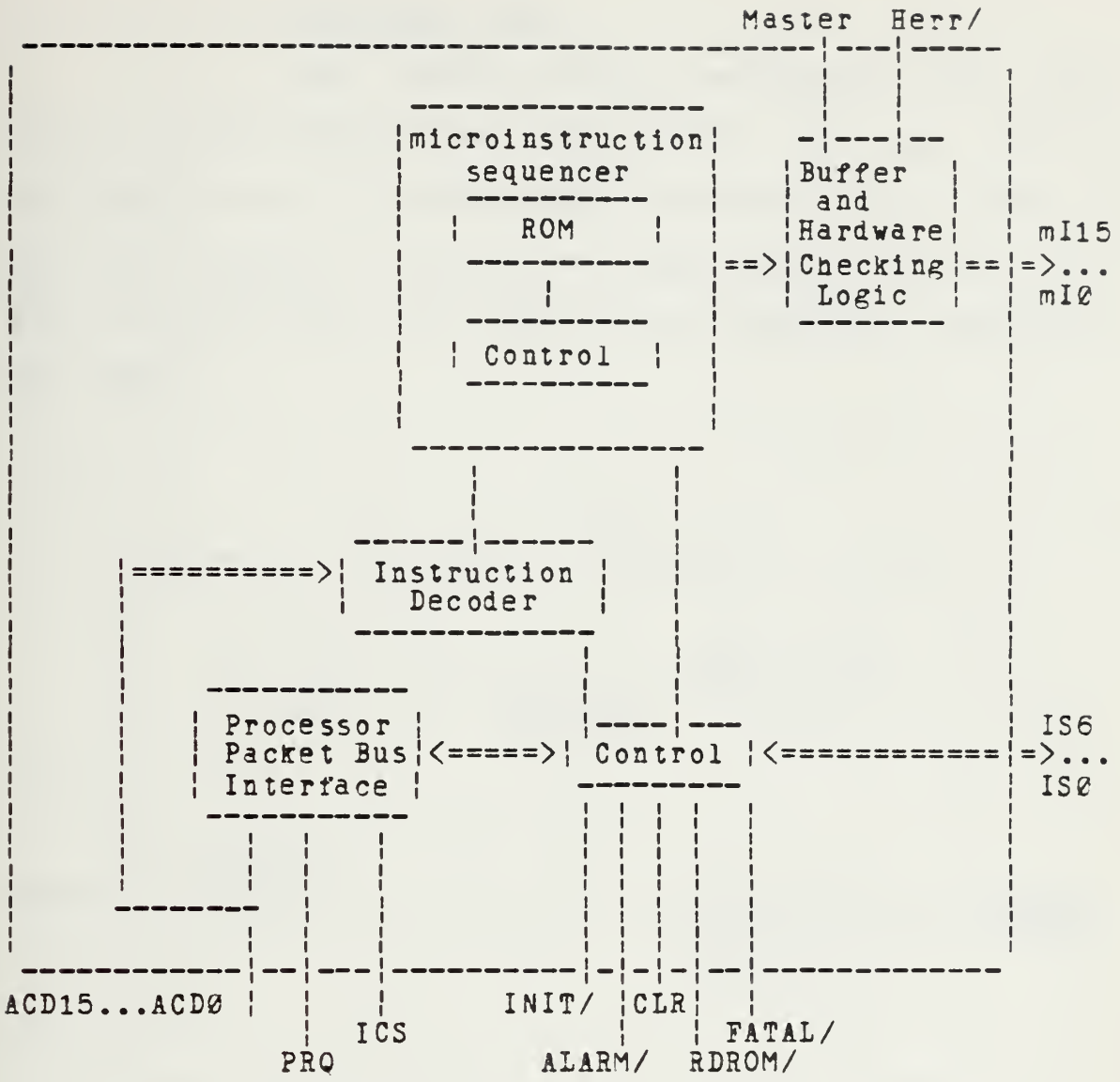


Figure 3.4 43201 Block Diagram.

b. Microinstruction Sequencer

The second subprocessor in the pipeline is the Microinstruction Sequencer (MS). The role of the Microinstruction Sequencer is to decide which microinstruction should be sent to the Execution Unit for each cycle.

c. Execution Unit

The 43202 contains the Execution Unit (EU) which is third stage of the GDP pipeline. This unit receives microinstructions from the 43201 and routes them to one of the two independent subprocessors that make up the EU, the Data Manipulation Unit (DMU) or the Reference Generation Unit (RGU).

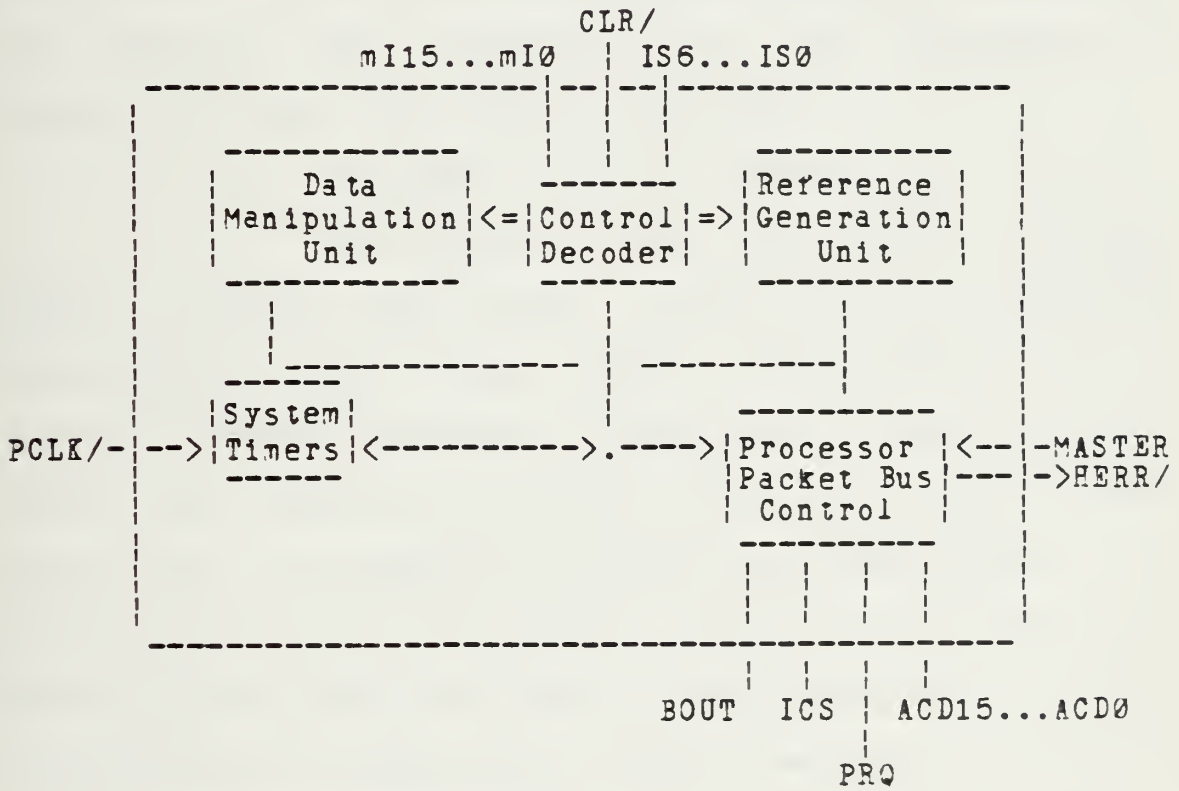


Figure 3.5 43202 Block Diagram.

The Data Manipulation Unit contains the registers and arithmetic capabilities necessary to perform hardware recognition of eight (8) data types, 16, and 32 bit multiply, divide, and remainder

operations. Additionally it contains the control functions for 32, 64, and 80 bit floating point arithmetic.

The Reference Generation Unit provides the translation of a 40 bit virtual address into a 24 bit physical address, a hardware enforced domain protection system (read, write, alter, accessed), handling sequencing for 8, 16, 32, 64, and 80 bit memory accesses, and controling the on-chip top-of-stack register. In general, the Execution Unit manipulates data and translates the logical addresses into physical addresses.

d. Processor Packet Bus Definition

The Processor Packet Bus contains 19 signal lines. Sixteen (16) signal lines are the three-state Address-Control-Data lines (ACD15 through ACD0) and the remaining three are control lines used to request the bus, enable the buffers for output from and an interconnect status line. In general by sending the proper signals, the system bus is used as an address, control or data bus. The mechanism for this multi-use of the system bus is quite complex and it is explained in Intel's manuals.

2. Interface Processor

The Interface Processor (IP) interfaces the GDP's and the employed peripheral subsystems by providing I/O facilities. An IP maps a portion of the peripheral address space into the iAPX 432 system memory via four (4) different IP windows that may be mapped onto four (4)

different objects located in main memory, with the same protection mecnanisms as any iAPX 432 processor. Figure 3.6 illustrates the internal architecture of the Interface Processor.

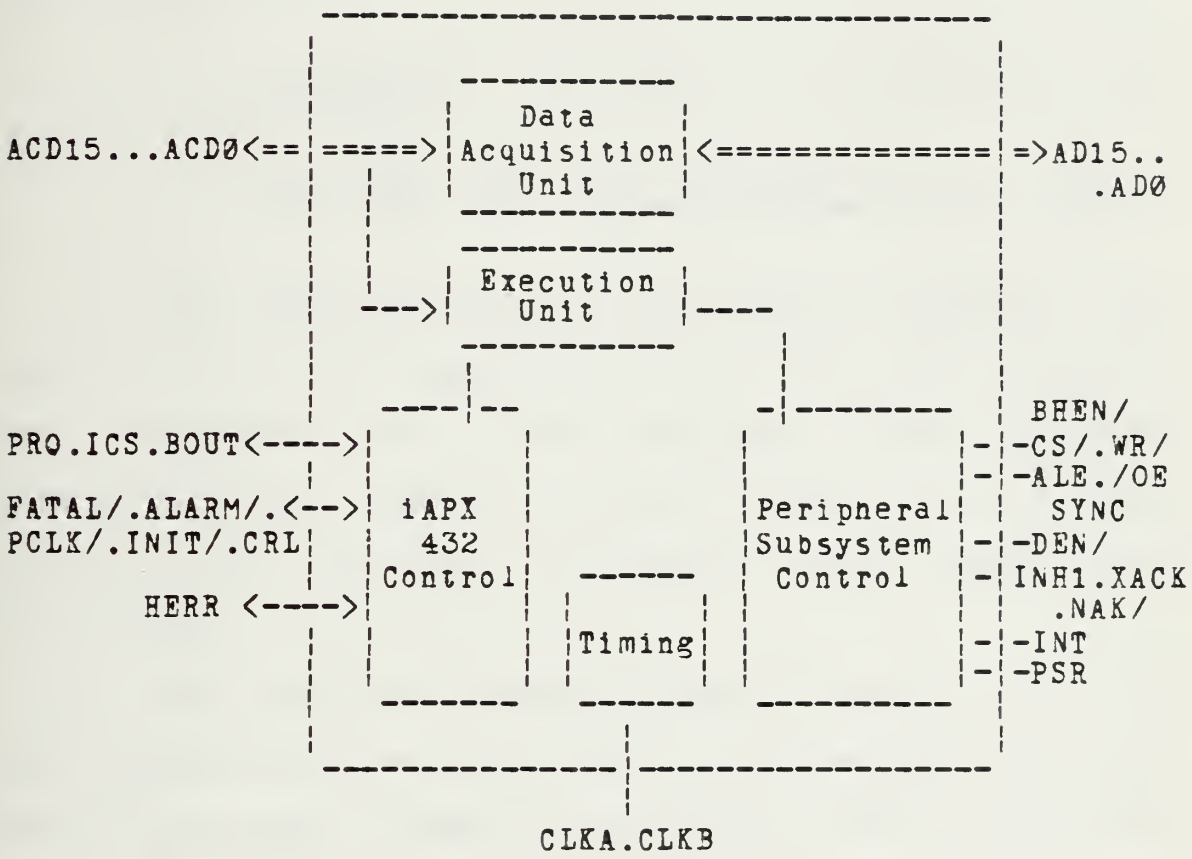


Figure 3.6 Interface Processor Block Diagram.

D. DATA MANIPULATION

1. Hardware Recognized Data Types

Computers store data in their memory in terms of zeros and ones. What these strings of bits represent depends upon the interpretation given to them by the computer. This means that two or more different pieces of information might

be represented by exactly the same string of bits and so this string could be interperated as an instruction, integer, or character. These basic representations are called hardware recognized data types and have the following characteristics:

1. Each instance of a type is an addressable unit of memory.
2. Each type has associated with it a set of operations that can be performed on this type.

The combination of all the hardware recognized data types with all the operators which act upon them constitute the instruction set of the computer. These hardware recognized data types are called primitive data types because they are used to construct more complex data structures.

The Intel 432 recognizes eight different primitive data types, which are divided into four classes: Characters, Ordinal (unsigned integers 16 and 32 bits long), Integer and Real. Figure 3.7 provides the formats of each of the eight hardware recognized data types [Ref. 15].

2. Structured Data Types

The term structured data types is used for ordered aggregates of primitives. The two structured data types that can be accessed by the 432 are Arrays and Records. Although not supported by hardware, the 432 provides a mechanism that allows structured data types to be manipulated easily.

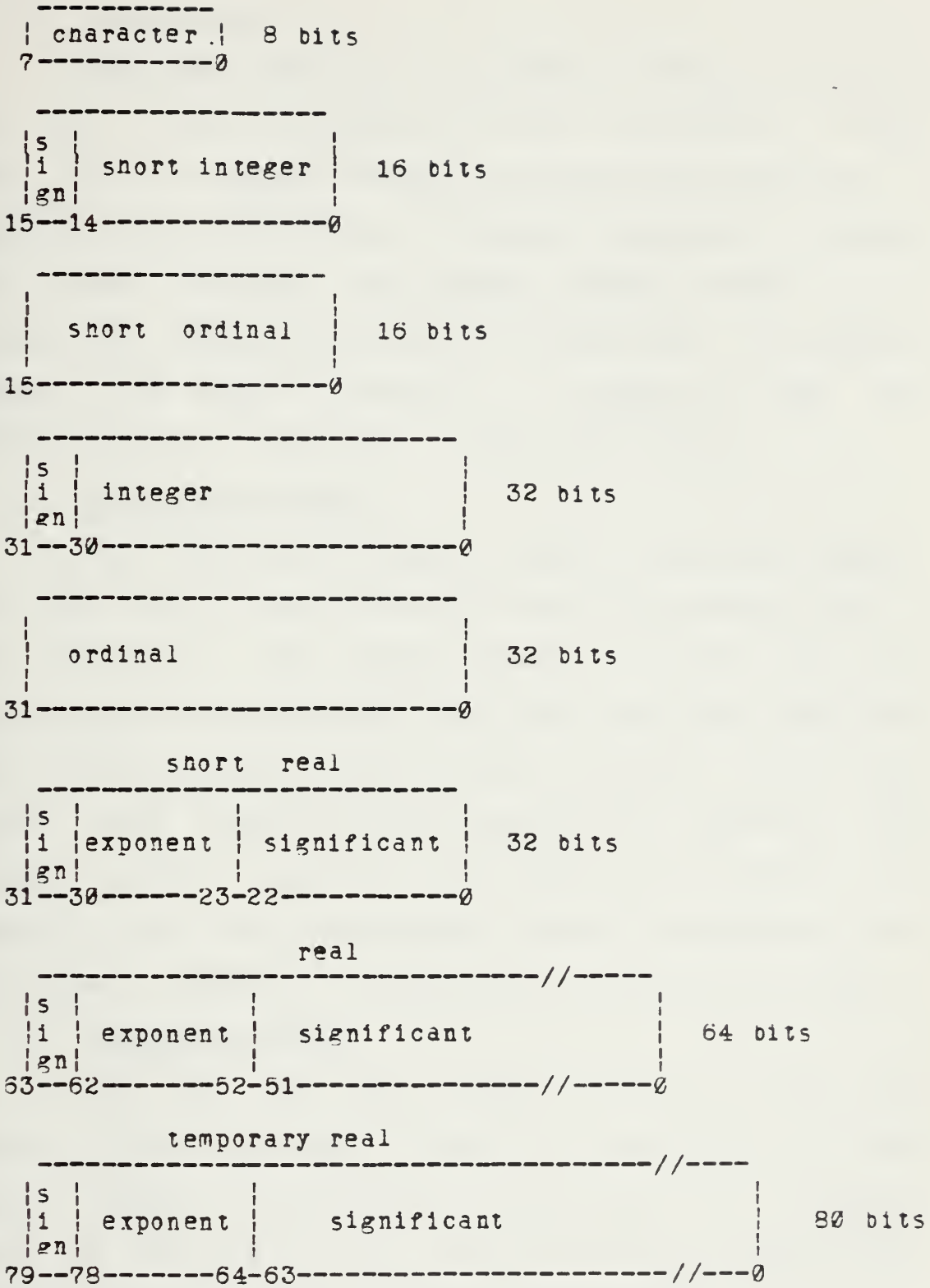


Figure 3.7 Primitive Data Types.

3. Instructions

The instruction set of the 432 is rich and supports five out of the six operations which, according to Myers [Ref. 10], a language directed architecure must provide. It supports string processing, arithmetic operations, program flow, error handling, and program tracing, while it is missing the editing operation. In addition, it supports segment and object creation. A summary of the 432 instruction set is contained in Myers [Ref. 11].

4. Instruction Format

The instruction format of the 432 has four fields. The first field (class) specifies how many operands are in the instruction. The second (format) specifies how the operands are to be accessed. The third (reference) specifies the logical addresses of up to three operands. Finally, the fourth field (operator code) specifies the operator. The segment selector identifies the segment that contains the operand, while the displacement locates the operand within the segment. Figure 3.8 illustrates the instuction format.

5. Addressing Modes

The base and index values are used to locate the operand within the segment either by containing the value itself (direct addressing) or by pointing to a location which in turn points to the location where the value of the operand has been stored (indirect addressing).

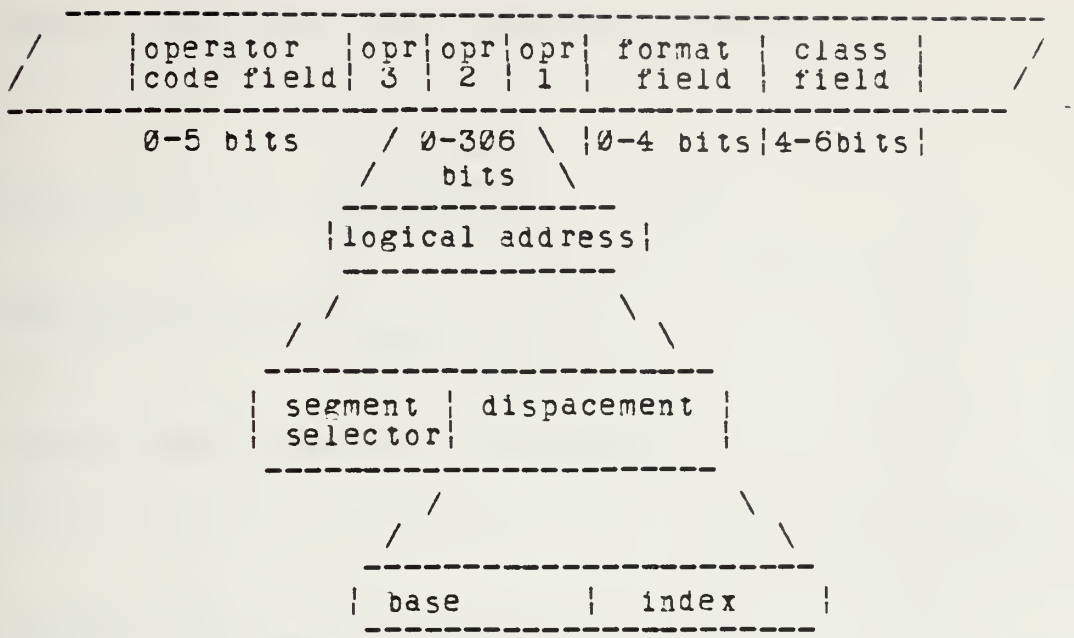


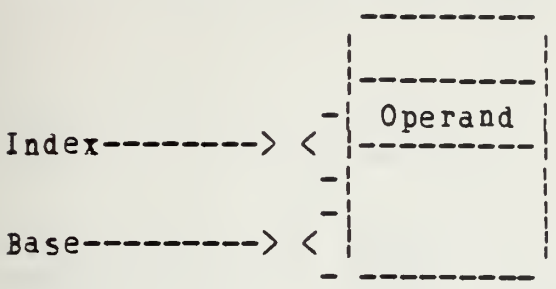
Figure 3.8 iAPX 432 Instruction Format (three operands)

A number of combinations (base direct/indirect and index direct/indirect) of direct and indirect reference are called addressing modes. The system selects the most efficient way for addressing the various data types as follows:

1. Base and Index Direct : used to access scalars.
2. Base Indirect, Index Direct : used to access records.
3. Base Direct, Index Indirect : used to access static arrays.
4. Base and Index Indirect : used to access dynamic arrays.

Figures 3.9A and 3.9B illustrate the four addressing modes.

Scalar: Index and Base Specified Directly



Record: Base Specified Indirectly

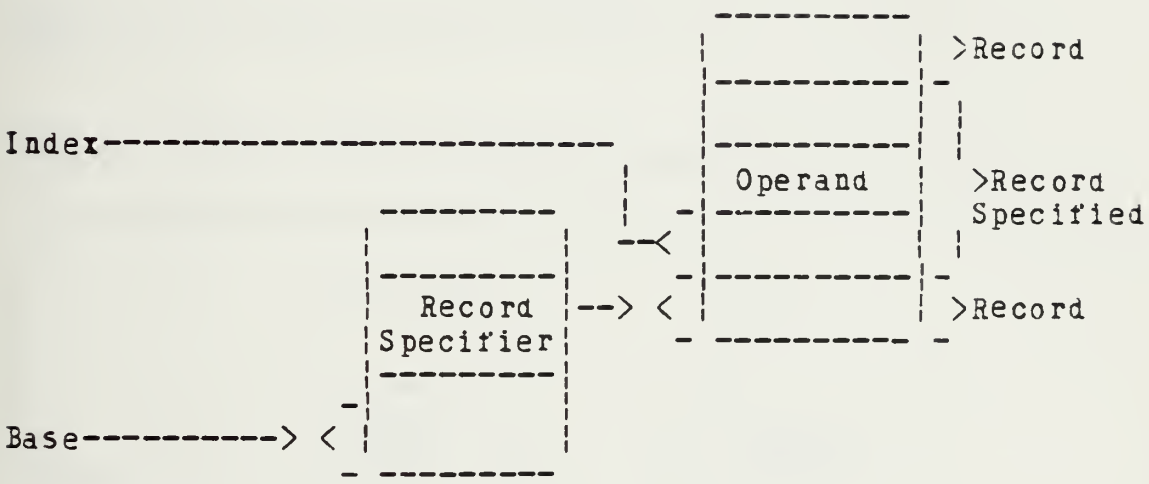
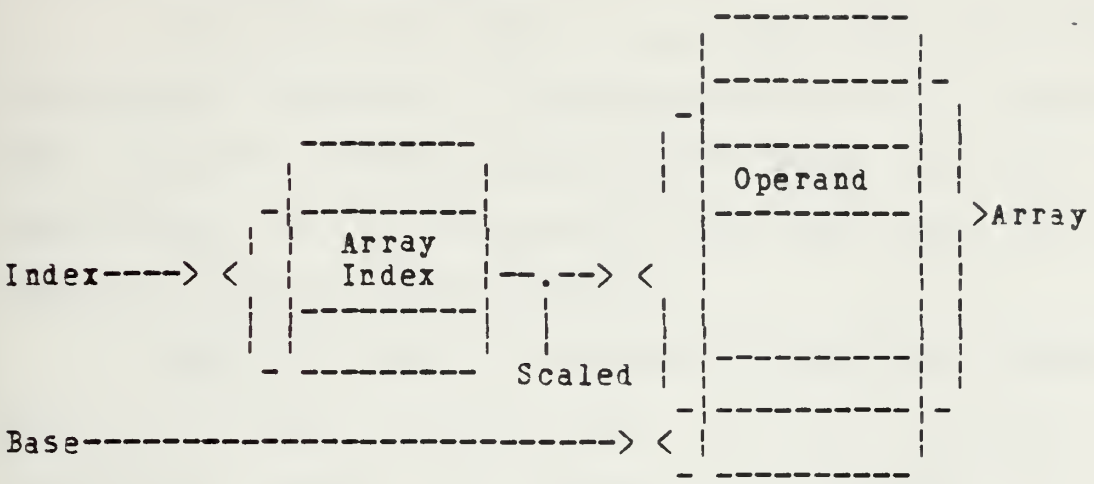


Figure 3.9A Addressing Modes

6. Operand Stack

A special data segment maintained by the hardware for expression evaluation is called the operand stack. Accesses to the top of the operand stack are called implicit references and the items are added to or removed from the stack on a last-in first-out (LIFO) policy. The current stack top is pointed to by a hardware maintained stack pointer.

Static Array: Index Specified Indirectly



Dynamic Array: Base and Index Specified Indirectly

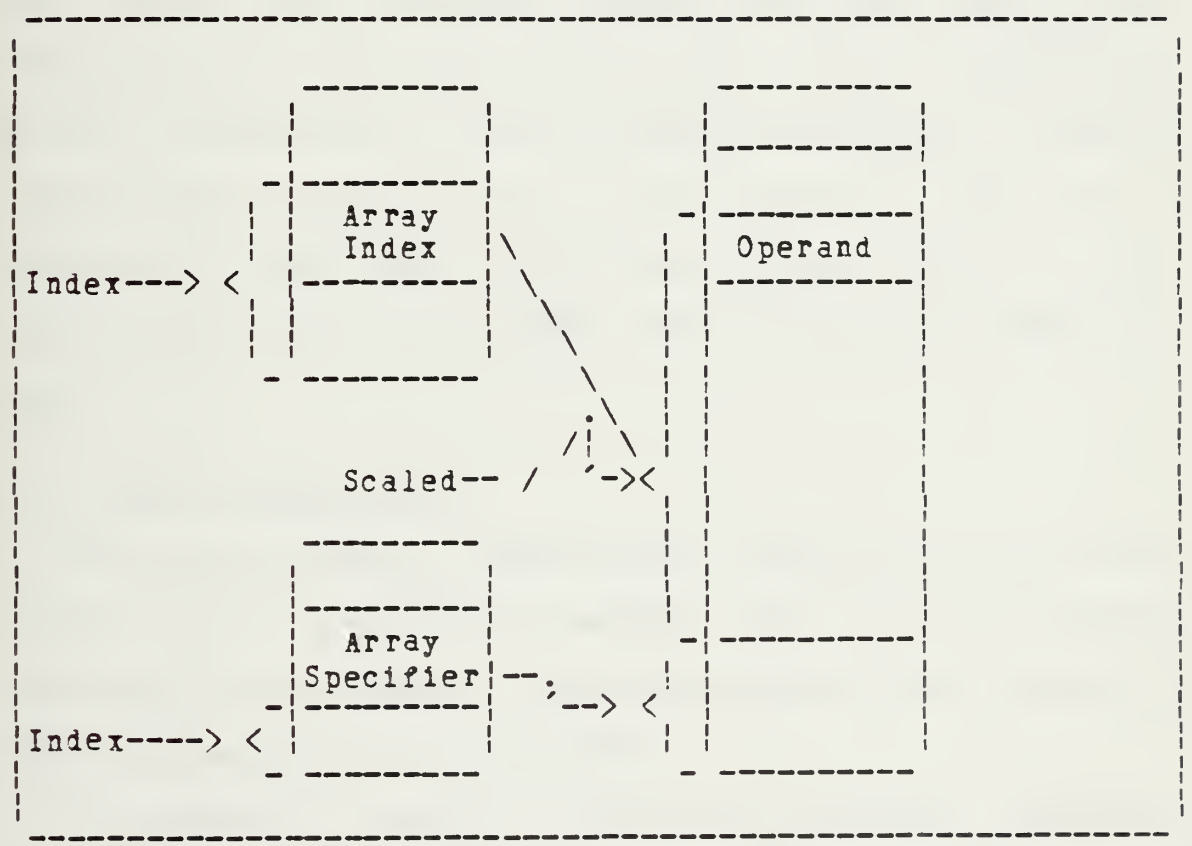


Figure 3.9B Addressing Modes

7. Instruction Encoding

An effort has been to speed up execution through instruction decoding. A large percentage of a computer's time is spent fetching instructions; hence the faster the instuctions can be fetched the less time will be spent waiting to execute an instruction.

The 432 uses bit-variable instructions versus the fixed-size length which is used by most machines. This flexibility implies that there is no constraint for the instructions to start or to end on byte or word boundaries. The size of the instructions varies from 6-bit long for the shortest up to 344 bits long for the longest instruction. Anotner difference, in the instruction encoding, is that in instructions like A = A + B, the operand A need to be referenced only once since the format field in the instruction indicates how many times a particular operand is used.

E. MEMORY ORGANIZATION

The term memory organization comprises two distinct aspects of a computer's memory. One is the hardware structure of the memory, while the other is the method by which the memory can be addressed.

In general all memory is structured linearly, differing only in the number of bits per byte and the way a byte is addressed. Memory is addressed through the address pins of the CPU, which are connected to the decoder, and translated

to addresses. Since there are no gaps in the sequence of numbers produced by the decoder, any memory is addressed linearly. However, if you look at memory from the users point of view, a memory is said to be:

1. Linear if the user can address it linearly.
2. Segmented if the user can address blocks of linear address, where each space is called a segment.

Each item within a segment is addressed by a two component address:

1. The segment selector which specifies the segment.
2. The displacement which specifies the offset from the base of the segment to the item being selected.

1. Mapping Segmented Memory

In general, mapping is performed via the segment discriptors that are maintained in the segment table. Each segment has a corresponding segment descriptor which contains the starting physical address and the length of the segment. This mapping is illustrated in Figure 3.10.

The segment selector of a logical address is used as an index to select a segment descriptor in the segment table. The displacement is added to the segment starting address to produce the physical address of the operand being referenced. Additionally, displacement is easily checked by the hardware to ensure that the reference does not exceed the length of the segment.

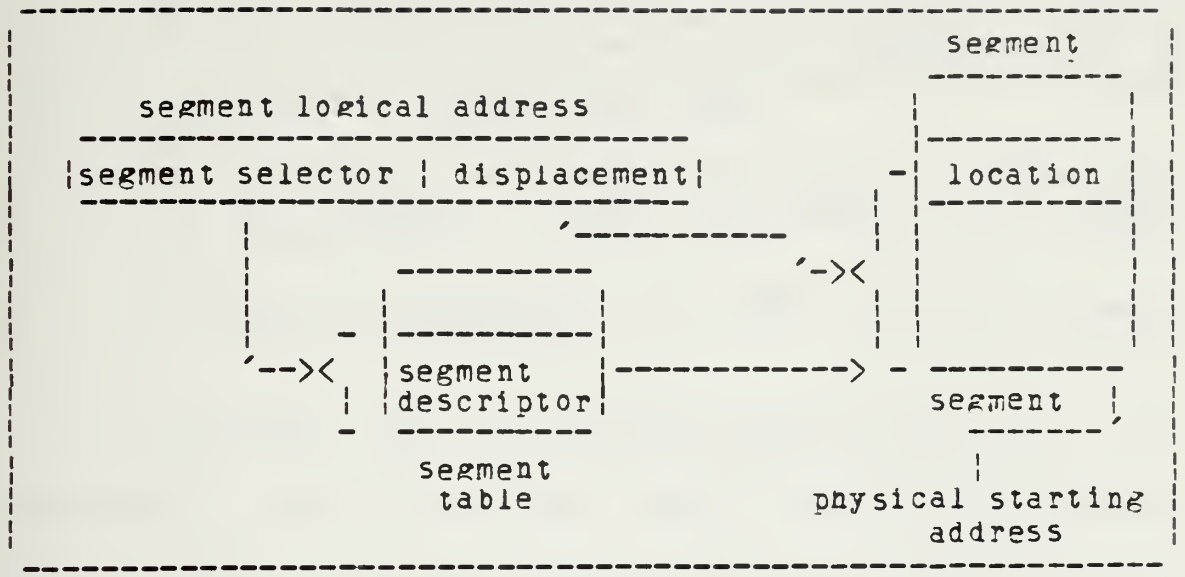


Figure 3.10 Segmented Mapped Memory.

2. Structured Memory

The Intel 432 has segmented memory which is different from the segmented memories of other microcomputers as follows:

1. The Intel 432 can address up to 2**24 segments. Each segment can be up to 2**16 bytes long. Therefore, the total virtual address space is 2**40 bytes.

2. The Intel 432 uses a two-step mapping process that separates segment relocation and access control.

For these reasons Intel refers to the 432's memory as structured memory.

Two-Step Mapping: The access mapping, being similar to one-step segment mapping process, is illustrated in Figure 3.11 and it is performed as follows:

1. The segment selector part of a logical address is used as an index into the access segment to select one of the access descriptors, which contain access rights data.

2. The access descriptor then acts as an index into the segment table to select a segment descriptor.

3. The displacement is added to the segment starting address.

Although there are several segment types, the two fundamental types, called base types, are the data segments which contain both data and instructions, and the access segments, which contain only access descriptors. Each base type is divided into several hardware recognized system .pa types. For example, data segments are divided into instruction segments and stack segments.

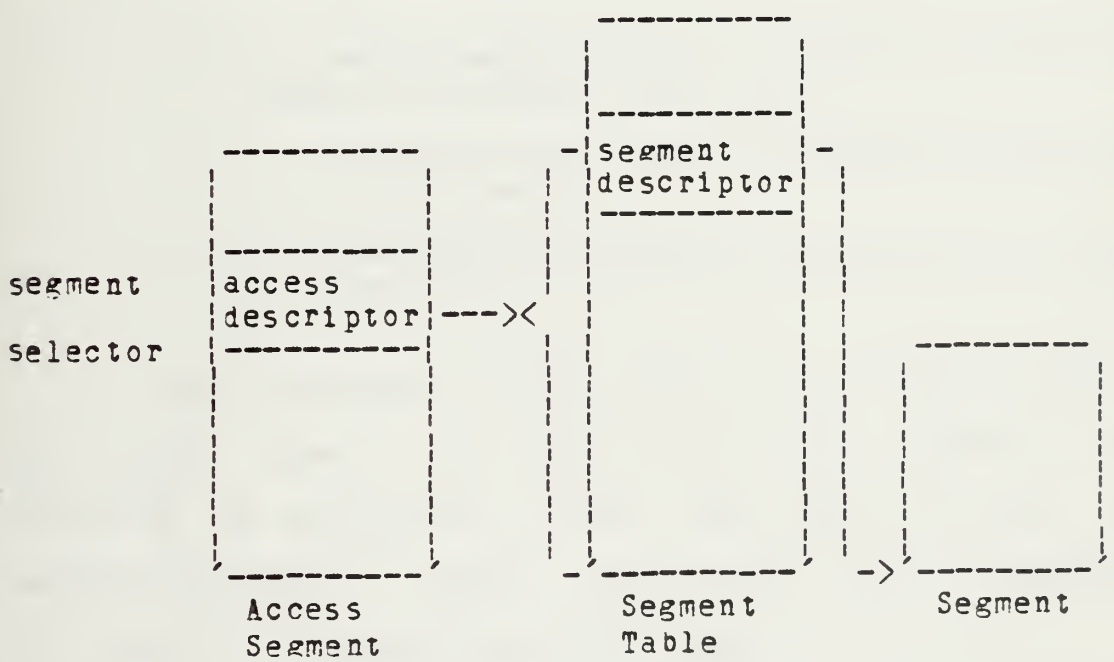


Figure 3.11 Two-Step Mapping.

Since two-step mapping process separates segment relocation from access control, the type information is contained in the segment descriptor while the access information is contained in the access descriptor. This organization allows the division of protection into user-specific protection (access rights) and segment-specific protection (segment type protection).

Each program module is supplied at run-time with a collection of segment numbers for only those segments it needs to access during execution. These segments determine the access environment and they are stored in a collection of access segments.

The advantages one can view in the two-step-mapping can be summarized as follows:

1. It takes fewer bits in an address to specify a segment (as few as four bits).
2. It restricts the number of segments accessible by a given program.
3. It separates the protection features into access and type protection.

One potential problem with the two-step mapping is the access time. To speed up the process, Intel maintains an associative cache that stores recently used segment descriptors, access descriptors, and the addresses of a number of commonly accessed items.

3. Object Referencing, Addressing and Protection

In either the single or the multiuser environment objects need protection from the user who might inadvertently use an object in a way that it was not intended to be used and from programs that might destroy the object. Additionally, in a multiuser environment when several users require simultaneous access to the same object, at least two more problems can arise:

1. Access Rights: Not all the users need to have the same access rights.

2. Exclusive access: One user may require exclusive access to that object for a particular operation to ensure that no other processor alters the object while the operation proceeds.

These problems are solved in the 432 architecture by the use of the access descriptors.

4. Access Descriptors

Each object has its own access descriptors which act as both priviledge and protection permits for the object. Many access descriptors can exist for the same object. Each access descriptor belongs to some (but may be copied and/or passed to another) procedure (context), and defines the access rights of the procedure using this object. A procedure can reference a segment of memory if and only if it holds an access descriptor for that segment. The basic rights accorded the procedure by the access descriptor can be read only, write only, both or none.

5. Segments versus Objects

An object can be a contiguous subsection of a segment or it can consist of a single segment, or collection of segments. Figure 3.12 illustrates the relation between segments and objects.

The term segment refers to the physical structure of the data in memory while the term object refers to the logical structure of the data in memory. Viewing an object as a single entity, its root segment specifies the beginning of the object while the descriptor segment that points to this root segment is considered as pointing to the whole object. Such a segment descriptor is called object .pa reference, while the segment table containing object references is called object table.

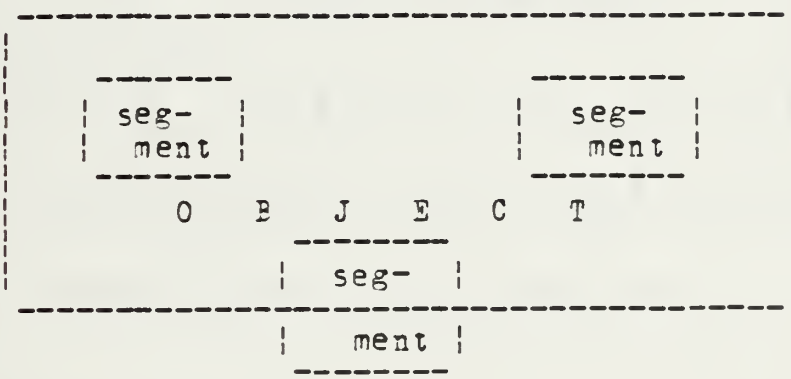


Figure 3.12 Relation among Objects and Segments.

F. OPERATING SYSTEM

The unique item in the 432 arcitecture is the way its operating system is implemented. There are two distinct,

not self-complete but cooperating, operating systems; one is implemented in hardware and is called the Silicon OS, while the other is a collection of Ada packages available to the user for further modification as required, and called the iMAX OS. Figure 3.13 illustrates the mechanisms implemented in the Silicon OS and the cooperating iMAX OS as well as the conventional architecture level and the user interface level.

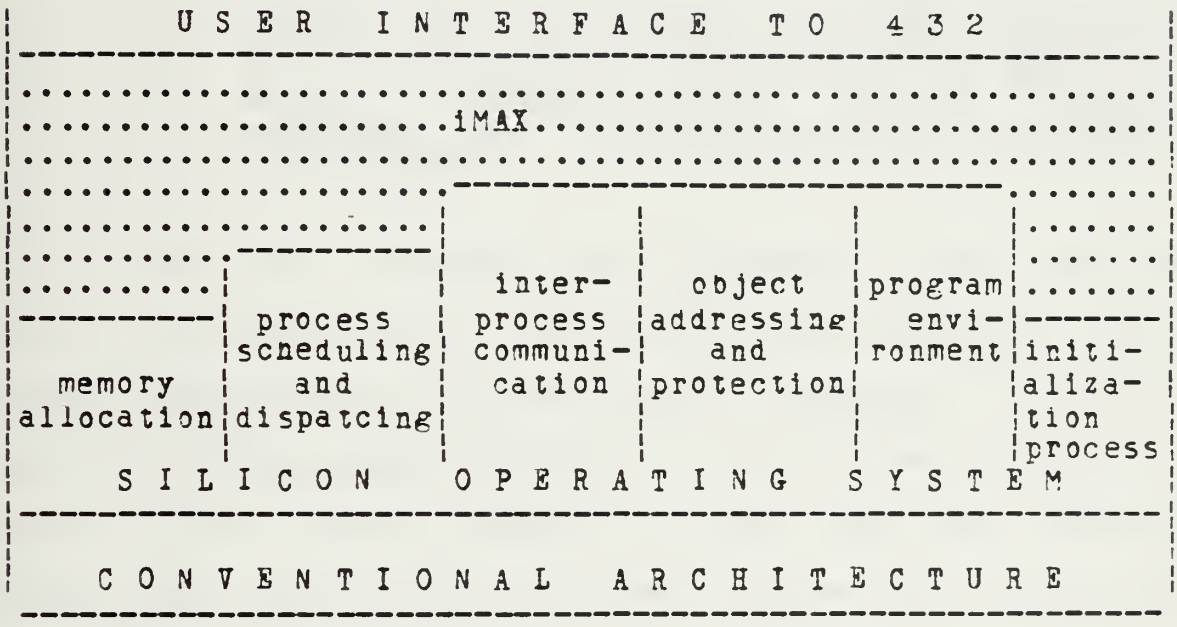


Figure 3.13 Silicon and iMAX OS's.

1. The Silicon OS

A number of resource management mechanisms are implemented in the hardware, while the resource managment policies are established by software in iMAX OS. These mechanisms are what Intel calls the Silicon OS, which consists of a memory resources manager and processor

resources manager. The mechanisms supported by the Silicon OS are illustrated in figure 3.13.

2. iMAX OS

The iMAX OS is a collection of Ada packages provided in two different subsystems. The users can select the one best tailored to their applications:

a. Full iMAX provides run-time support including dynamic storage management, dynamic process management, and I/O interface.

b. Minimal iMAX is a subset of full iMAX and supports the execution of multiple processes on a multiprocessor environment. It does not support dynamic storage and/or process management and I/O except through the DEBUG-432 debugger.

The only compiler that generates code for the 432 is the Ada compiler provided by Intel. Although this forces programmers to use only the Ada language for writing programs running under the 432, it allows them to make easy calls to the iMAX packages by using the Ada provided facilities, such as the environment pragma, with clause and use clause.

Two kinds of processes can be controlled by iMAX; the static processes which are created by the user at system initialization and the dynamic processes which are created dynamically by the system and then entered into the ready state. The minimal iMAX, which does not support dynamic operations, requires only 47K of memory compared to 280K for the full iMAX.

The iMAX packages manage and enhance the 432's interprocess communication, data transfers between peripheral devices and the 432 central system, and allows the users to configure and control a number of General Data Processors and Interface Processors, static user processes, and I/O device interfaces.

G. PROGRAM STRUCTURE

In most computer systems program structure is part of the operating system or the language run-time environment and there is no need for the program structure to be included in the description of the architecture. However, in the Intel 432, the program structure is part of the architecture and contains the following objects:

1. Processor Objects
2. Process Objects
3. Context Objects
4. Instruction Objects
5. Data Objects

1. Processor Object

A previous section specified that the processors in a multiprocessor environment are hardware-recognized objects and that object is a data structure in memory. It is logical therefore to ask how can a GDP, made of silicon and sitting on a pc board, where it fetches and executes instructions, be a data structure in memory?

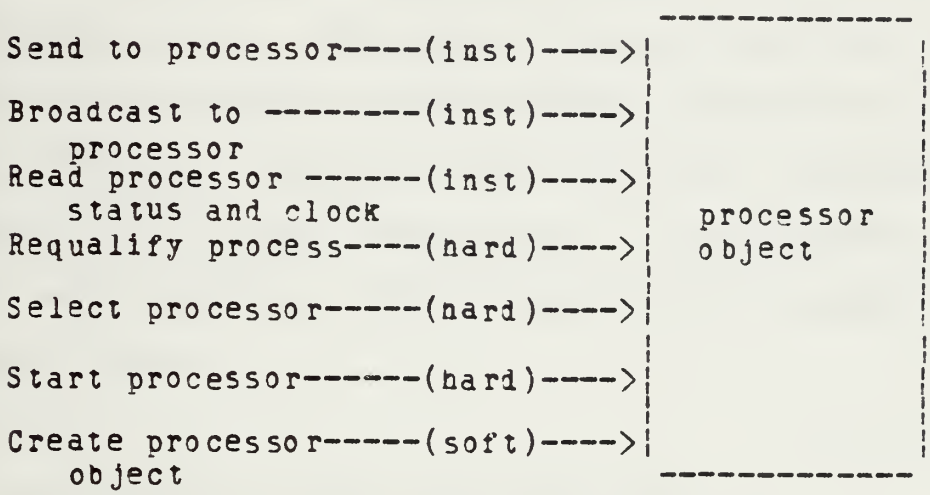
The physical processor is not an object. However, each physical processor does have an object (a data

structure in memory) associated with it which is called a processor object. The processor object is an abstract representation of the GDP and is in one-to-one correspondence with the GDP. It can reside anywhere in the memory and can be dynamically relocated if necessary. Addressed by its associated processor via an on-chip reference, the processor object provides the means to assign the processor it represents to a particular process set. It also serves to record information about the physical processor such as its state (running or halted), diagnostic information and references for other objects the processor needs. Additionally, it contains an object reference for the process object currently running on the processor. This reference changes in time as the processor switches among different processes. The processor object is depicted in Figure 3.14 with the operations which act upon it.

Remember that the GDP is a three stage pipeline processor which consists of two chips, each one of which has two subprocessors. If the programmer tries to keep all of these details in mind as well as the flow of information and the data manipulation, while writing a program statement, complexity and confusion will reign. However, if the programmer considers that a processor exists which can perform a set of operations, programming is simpler.

2. Process Objects

A process is a locus of control within an instruction sequence. That is, a process is the abstract



Legend: inst = operation is a machine instruction (operator).

hard = operation is performed by hardware as a result of conditions detected by the processor.

soft = operation may be implemented by software (e.g. an operating system).

Figure 3.14 Processor Object.

entity which moves through the instructions of a procedure as the procedure is executed by the processor. Given this definition for a process object, how can a sequence of operations be an object? The process is not an object, but each process does have an object associated with it which is called a process object. The process object contains information concerning how the process should be scheduled and what is the process status (running, waiting, etc.).

The significance of the existence and the use of the object reference is that the 432's object oriented addressing and protection mechanism does not allow a processor to access an object unless it has an object reference for it. The object reference from the processor object to the process object changes dynamically in time as the processor switches among different processes. Each process has its own process object. The processes may share the same processor. Figure 3.15 indicates operations on a process object.

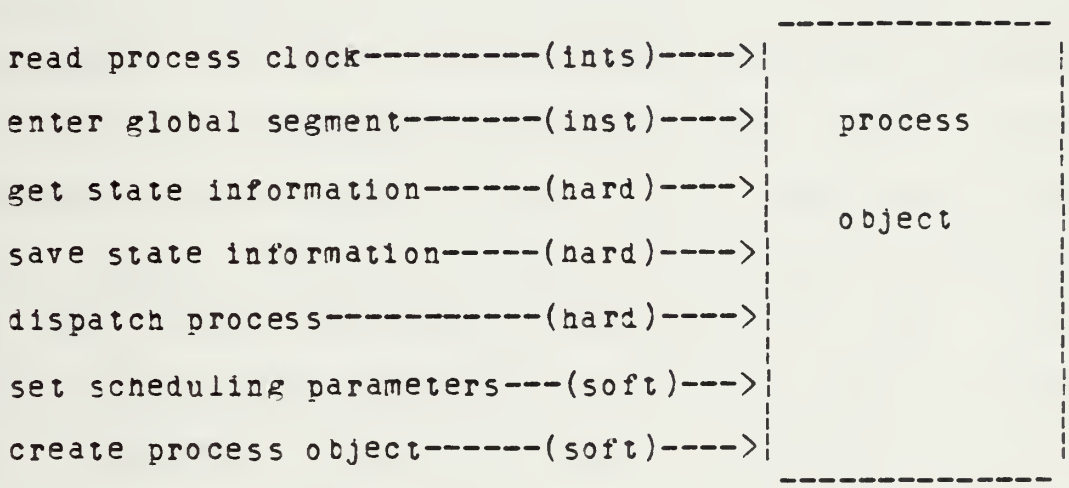


Figure 3.15 Process Object.

3. Context Objects

Procedures (e.g. SQUAREROOT X) are shared among several different processes by giving each process a copy (an instance) of the procedure. Each instance of a procedure (each copy) is called a context.

A context object contains information about a particular invocation of a procedure. This means that there are as many context objects for the same procedure as there are invocations of the procedure. The information that a context contains includes the instruction pointer for this context, the stack pointer for this context's stack, the return link to the calling context, and references for all the objects that can be accessed by this context.

Figure 3.16 depicts a context object and the operations which act upon it. The complete list of objects currently accessible by the procedure is called its access environment. The 432's object-oriented protection mechanism does not allow a program to access an object unless it has a reference for that object. Thus in the context one can only access objects that are listed in the access environment, and the access environment changes only if the current context calls another procedure because the called procedure has a different context and hence a different list of objects that can be accessed. This places the protected access environment at the procedure level, instead of the process or job level as on most systems.

4. Instruction Object

An instruction object has two major characteristics:

1. It can only hold instructions (and a little bit of system information) and never data.

2. It is the only type of object that a processor will use as a source of instructions to be fetched and executed.

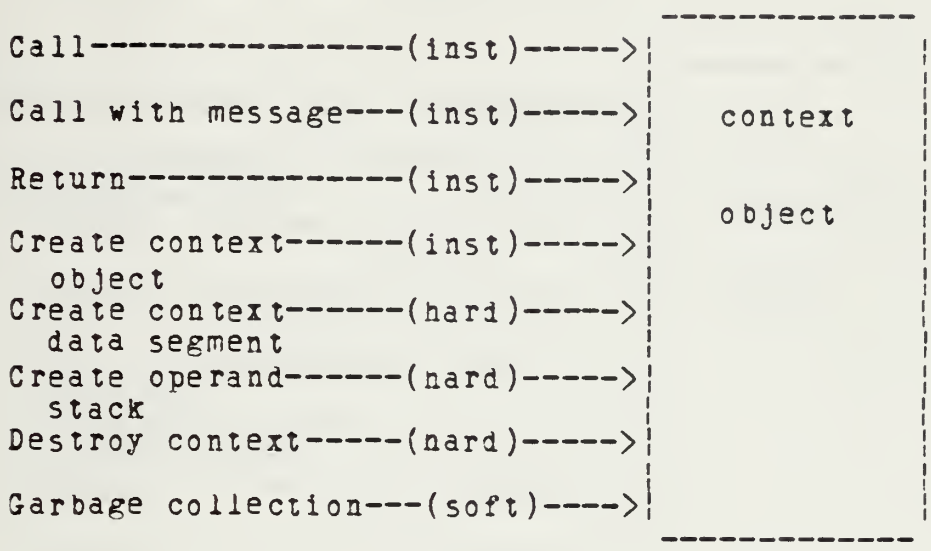


Figure 3.16 Context Object.

This second feature implies that the 432's object-oriented protection mechanism will never allow any other kind of object (process object, data object, etc.) to be accidentally mistaken for instruction and thus executed. This isolation of the instructions provides the advantage that all 432 programs are fully reentrant.

Instruction objects can be shared by several instances of a program. Figure 3.17 indicates an instruction object and the operations which act upon it.

5. Data Objects

Data object is an object that holds data (e.g. integers, reals, character, tables, etc.). The 432 provides a mechanism for assigning data objects (as well as other

objects) a software-defined type. This is another protection mechanism which ensures that the object will be manipulated by instructions of the proper type.

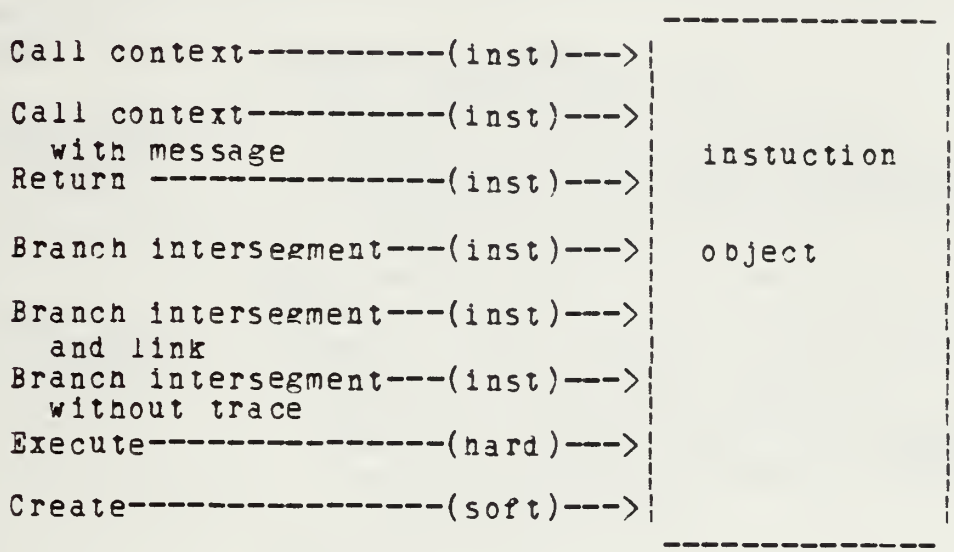


Figure 3.17 Instruction Object.

6. Summary of the Structure of a Program

Figure 3.18 summarizes the important points about each object of the program structure.

H. INTERPROCESS COMMUNICATION

It has been said that one of the salient characteristics of the 432 is its capability to support multiprocessing in a multiprocessor environment. In other words the system provides for concurrent programming. The mechanisms that support concurrent programming are quite complex and include the use of Communication Ports Objects, Domain Objects, and Dispatching Port Objects. Following an explanation of these

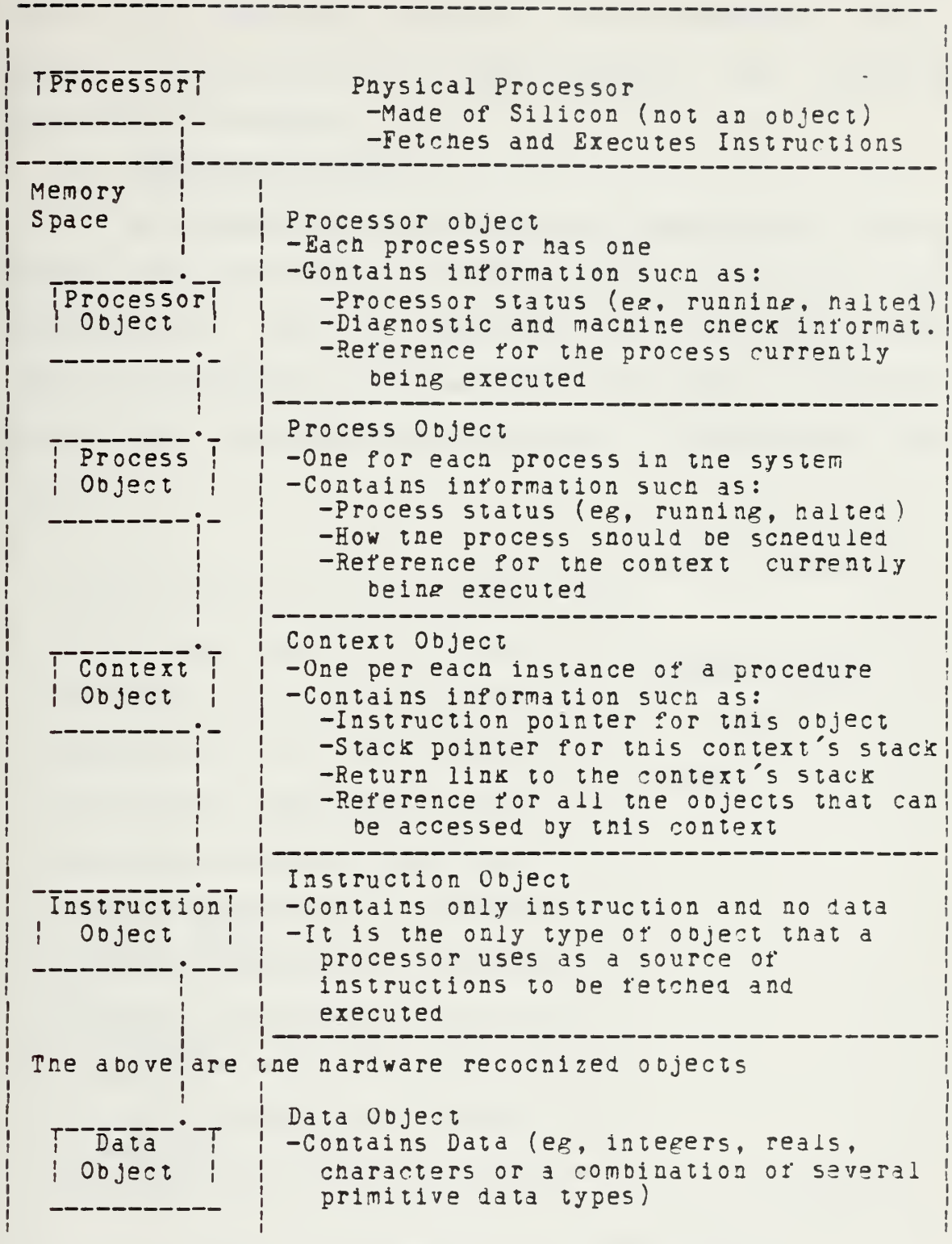


Figure 3.18 Summary of the Program Structure.

objects, a simplified example will be presented for a general understanding of the mechanism that supports concurrent programming.

1. Communication Port

A communication port is the communication medium between two asynchronously communicating processes. Each communication port is represented by an object which contains information on messages that are sent to processes. Figure 3.19 indicates operations on a communication port object.

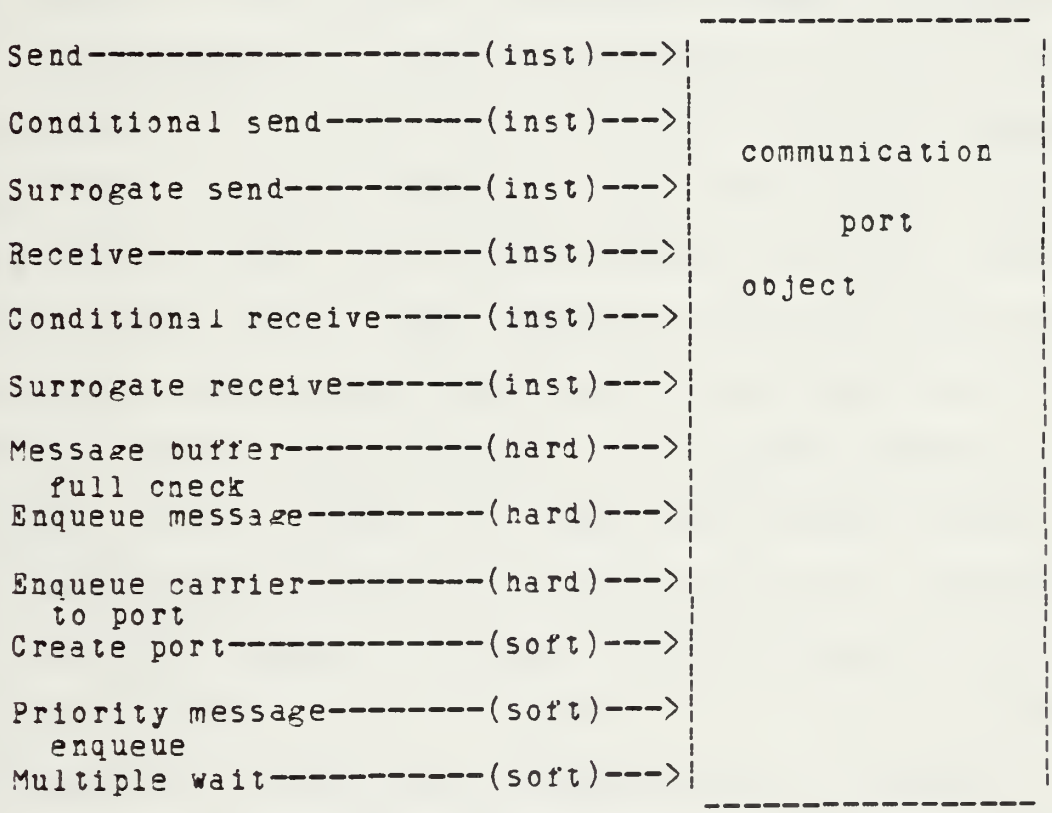


Figure 3.19 Communication Port Object

The communication ports, by acting as a buffer between two asynchronous processes, allow each process to proceed at its own rate.

2. Domain Objects

Imbedded in the features of Domain Objects are the concepts of static and dynamic objects. The concept of dynamic objects has been already encountered in Section G.3, where the context objects are examined. Recall that the access environment consists of objects for which an object reference exists in the list of the current context object. Additionally, the access environment changes dynamically when a procedure is called by a context and the access environment is now the list of the new current context. Since the objects accessible by the process change dynamically in time, the objects pointed to by the current context object vary in time. Therefore the space they occupy in memory is deallocated once they are not accessible by the proccess. Hence these objects are dynamic objects. In contrast static objects are those whose memory space is never deallocated regardless of whether they are accessible or not. In summary, a context object is also a list of dynamic objects while a domain is a list of object references for all the static objects used by a module.

In the object-oriented decomposition method, the criterion for modularization is for each module to hide a design decision. In order to carry out its task, a module

might consist of more than one procedure which in turn might need to access some, all or none of the static objects used by the module. This is the reason a domain contains a list of object references to all static objects of the module.

The reason objects associated with the modules are called domains is that the module they represent confine knowledge of the object structure within a specified domain. Figure 3.20 depicts a domain object and the operations which act upon it.

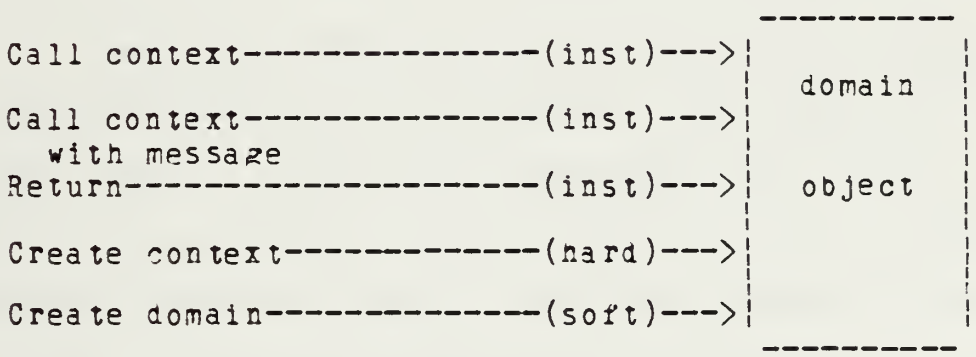


Figure 3.20 Domain Object.

Networks of domains, where each domain hides the representation of an object, represent the static structure of a 432 program. Some of the domains make use of objects in other domains in the network to create a more complex object.

If a module (represented by a domain object) needs to use procedures provided by another module, the first module must have a reference for the domain that represents the second module. The domains can be viewed

either from an outside or inside point of view. Consequently procedures can be executed inside or outside of the domain, this prevents static objects of the domain from being accessed by all procedures that use this domain.

In this case the static objects are divided as follows:

1. Private objects - accessible only by procedures inside the domain.
2. Public objects - accessible by procedures both inside and outside the domain.

Let us assume the situation where a procedure (of a module) is currently executing. This procedure uses the same domain object which represents the module that belongs to the procedure. In this case we say the procedure is executing inside the domain. The same procedure needs to make a call to another procedure not belonging to its module. For this to be achieved the first module must have a reference to the domain representing the module of the callee. After this object reference is established the caller can use the static objects only in the public domain of the callee. In this case the procedure is executing outside the domain.

3. Dispatching Port Objects

The dispatching port object consist of two queues that serve to match processes with processors. It is the object that is used to implement transparent

multiprocessing. Figure 3.21 Indicates a dispatching port object and the operations which act upon it.

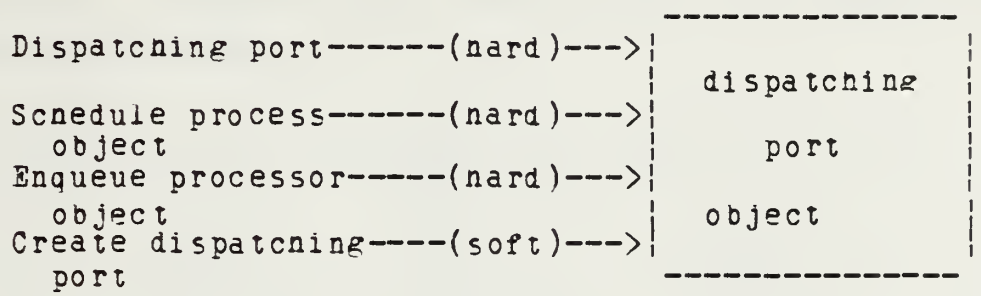


Figure 3.21 Dispatching Port Object

4. Example of Interprocess Communication

Consider a program, CHAT, which consists of two subprograms TED and JOHN. Program TED writes a report and then sends it to program JOHN to type it which in turn sends it back to TED for proofreading. When the report is correct TED prints it.

The subprograms TED and JOHN will be the two processes of CHAT which can start execution independently in two different processors of the system. For simplicity let us consider that TED starts execution first and begins to write its report. When it is finished, JOHN has been executing and it is ready to recieve TED's report for typing.

The mechanism for sending the report from TED to JOHN is the communication port which acts as a buffer between the two asynchronous processes, allowing each to proceed at its own rate. Two communications ports are needed

for our example. One is TED's receiver and JOHN's transmitter while the other is JOHN's receiver and TED'S transmitter. Figure 3.22 illustrates the communication between TED and JOHN.

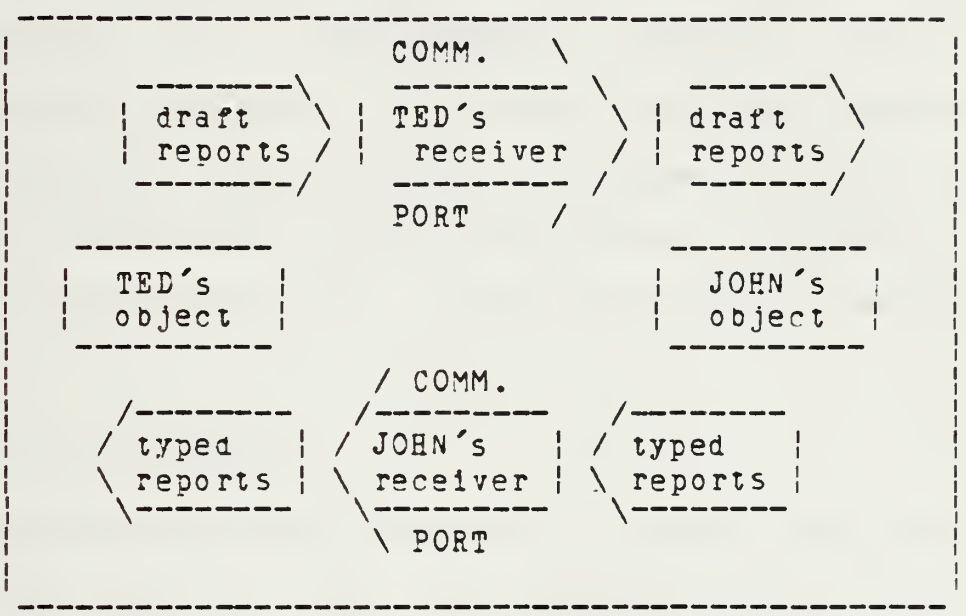


Figure 3.22 Interprocess Communication

The processes TED and JOHN interchange messages via the communication ports and the operations SEND and RECEIVE defined on the communication port object bear the burden of that intercommunication.

The instruction SEND has two operands. One is the message itself and the other is the communication port where the message is being sent. Before the SEND is executed the process has a message it wants to send. After SEND is executed, the message has been sent to the specified

communication port. For efficiency, only the object reference is copied.

The RECEIVE instruction has one operand, the communication port where the message is to be received. Before RECEIVE is executed there is a message waiting in the communication port. After RECEIVE is executed, the message is moved from the port to the process executing the RECEIVE. Again only the object reference is copied. In the case a RECEIVE instruction is executed without a message in the communication port, the process waits until a message is sent.

I. A BASIC I/O MODEL

Another complicated mechanism in the 432 architecure is the I/O operation. Here we examine the basic means for performing an I/O operation as well as the very basic implementation mechanisms needed for I/O operations.

A peripheral subsystem is an autonomous computer system with its own local memory, I/O devices and controllers, at least one processor, and software. The number of peripheral subsystems employed in any given application may be varied with changing needs, independent of the number of GDPs in the system.

To support the transfer of data through the wall that separates a peripheral subsystem from the main system, the Interface Processor (IP) provides a set of software-controlled windows. A window is used to expose a single

object in main system memory so that its contents may be transferred to or from the peripheral subsystem.

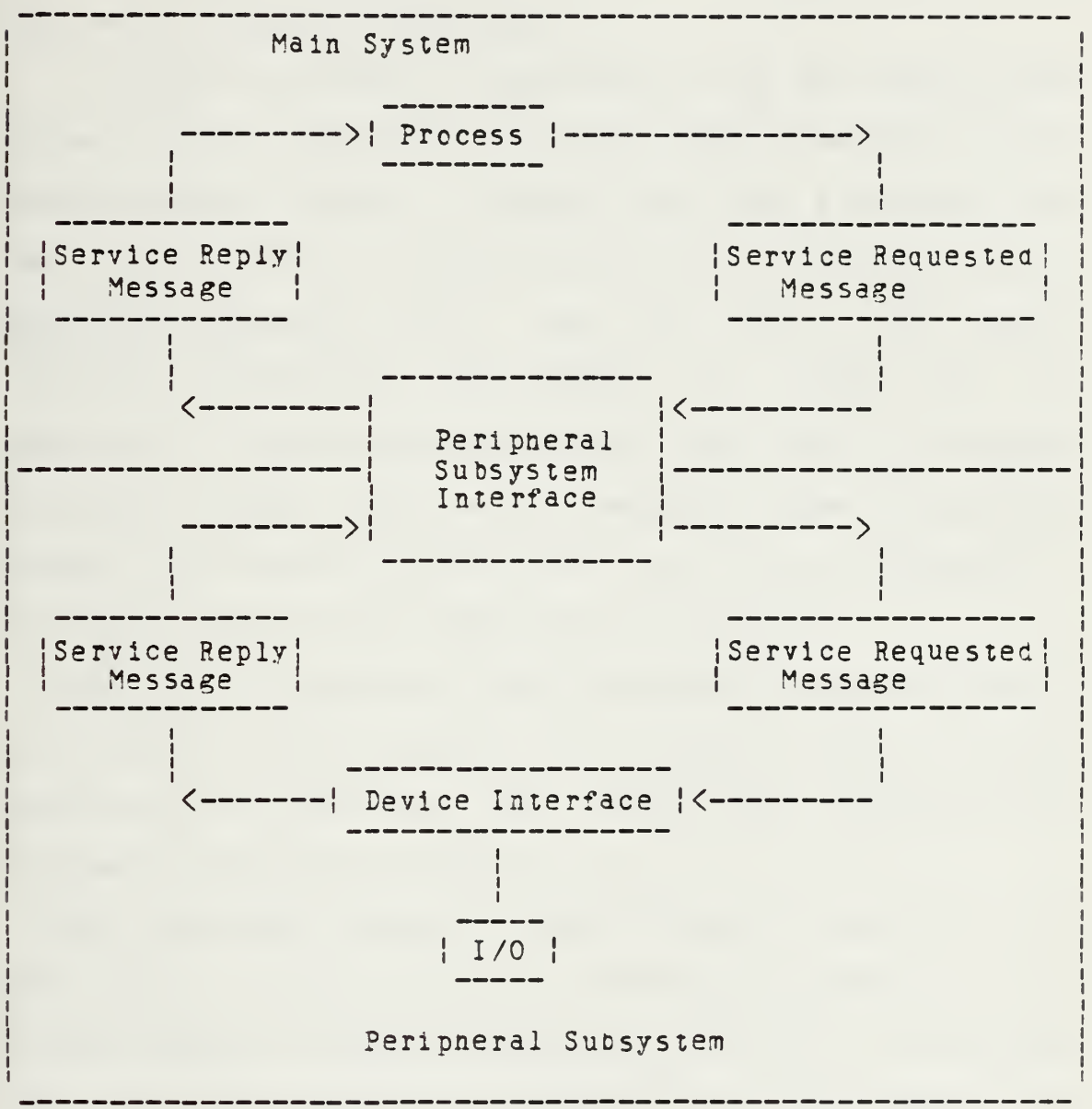


Figure 3.23 Peripheral Subsystem Interface.

The IP also provides a set of functions that are invoked by software. They generally permit objects in main system memory to be manipulated as entities, and enable

communication between main system processes and software executing in a peripheral subsystem. Figure 3.23 shows the peripheral subsystem interface.

A device interface is considered to be the hardware and software in the peripheral subsystem that is responsible for managing an I/O device. A message sent from a process that needs an I/O service contains information that describes the requested operation (e.g. "read file XYZ"). The device interface interprets the message and carries out the operation. If an operation requests input data, the device interface returns the data as a message to the originating process. The device interface may also return a message to positively acknowledge completion of a request.

The IP is connected to the peripheral subsystem bus as if it were a memory component; it occupies a block of memory addresses up to 64K bytes long. Figure 3.24 illustrates the peripheral subsystem hardware.

The Attached Processor (AP) and the IP interact with each other by means of address references generated by the AP and interrupt requests generated by the IP. At any given time, the IP controller is represented in main memory by a process object and a context object. Like a GDP, the IP itself is represented by a processor object.

When supporting multiple process environments, the IP controller selects the environment in which a function is to

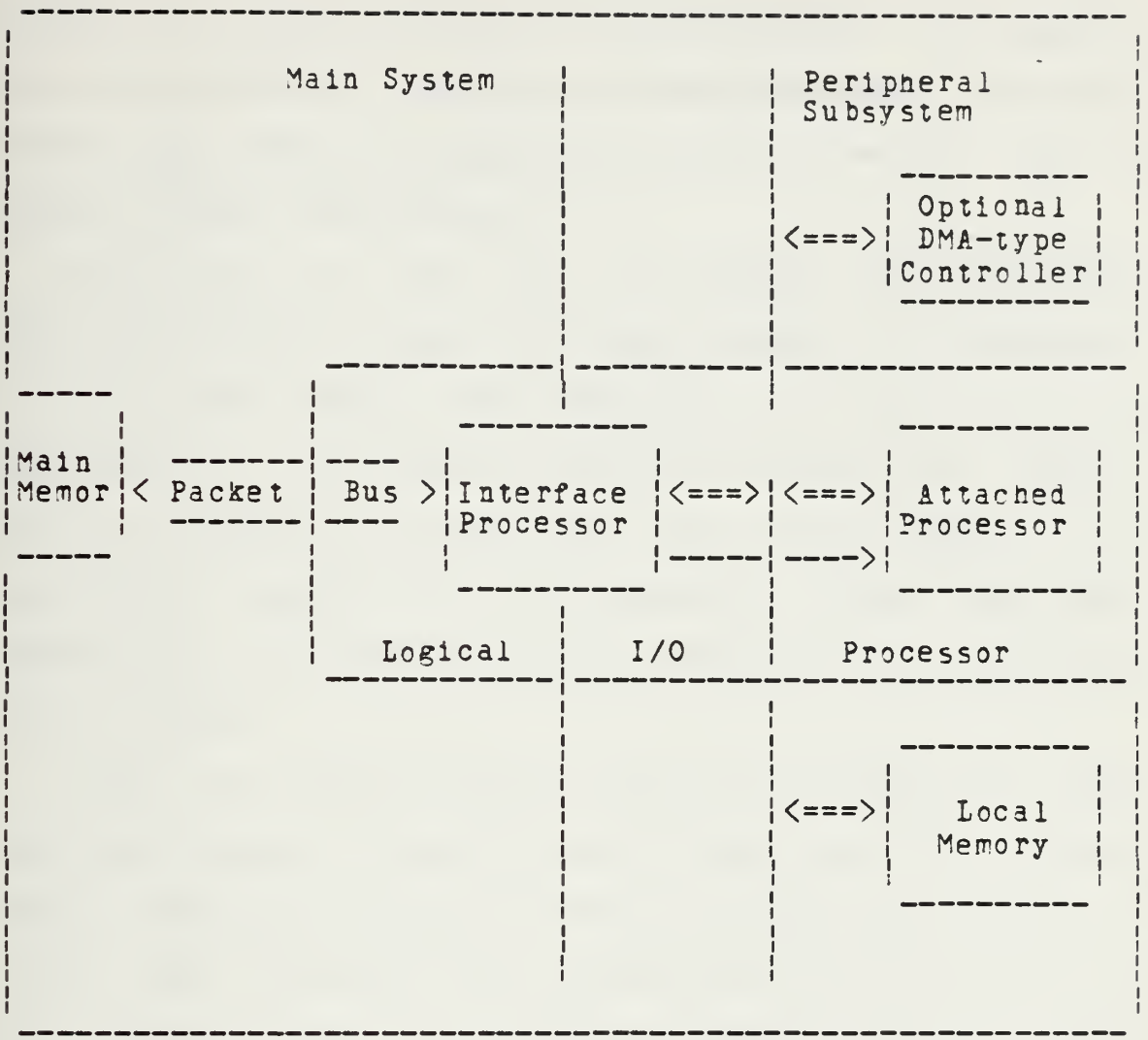


Figure 3.24 Peripheral Subsystem Hardware

be executed. If an error occurs while the IP controller is executing a function on behalf of one device interface, that error is confined to the associated process, and processes associated with other device interfaces are not affected.

K. WINDOWS

Every transfer of data between the main system and a peripheral subsystem is performed with the aid of an IP

window. A window defines a correspondence, or mapping, between a subrange of peripheral subsystem memory addresses within the range of addresses occupied by the IP and an object in main system memory.

The action of the IP, in mapping the peripheral subsystem address to the main system object, is transparent to the agent making the reference; as far as it is concerned, it is simply reading or writing local memory. Since a window is referenced like local memory, any individual transfer may be between an object and local memory, an object and a processor register, or an object and an I/O device.

There are four IP windows that may be mapped into four different objects. The IP controller may alter the windows during execution to map different subranges and objects. A fifth window provides the IP controller with access to the IP's function facility. By writing operands and an opcode into predefined locations in this window's subrange, the IP controller requests the IP to execute a function on its behalf. Upon completion of the function, the IP provides status information that the IP controller can read through the window.

Since there is a finite number of windows, most applications will need to manage them as scarce resources that will not always be instantly available. This means that

at least some I/O device transfers will have to be buffered in local memory until a window becomes available.

L. TRANSPARENT MULTIPROCESSING

The Intel 432 is especially designed to support parallel execution of programs by multiple processors. Absolutely no software changes are required to move programs from single processor systems to multiple processor systems or vice versa.

The first advantage of transparent multiprocessing is flexibility provided by a machine that offers a range of performance. Processors can be added to or removed from the configuration to meet the desired performance or price. The second advantage is reliability, because if one processor fails, the rest of the system may be able to continue running. In fact, during a benchmark performance test, one slot in the system motherboard prevented an assigned GDP from being initialized and the system continued to execute.

1. The Three Stages of Processor Management

The effective management of multiprocessing in a multiprocessor environment requires Policy Making, Scheduling, and Dispatching.

a. Policy Making

Policy Making establishes the criteria that determines how processes share processors. These criteria are defined by the user, via the iMAX OS, at system's

initialization. The iMAX OS passes this information to the Silicon OS for implementing the policy mechanism.

Options that can be selected to implement processor sharing are first - come - first - serve (FIFO), round robin, priority, and deadline weighted. The user can select the one which meets the applications requirements.

The Policy Making requires the user to set the scheduling parameters by answering questions such as: By when must the process be completed? How long does a process execute on the processor? How many turns can a process have before terminating?

b. Scheduling

Scheduling is the ordering of processes to run on processors in a manner that realizes the policy. It is part of the mechanism that carries out the policy.

A process is scheduled by sending the process to the dispatching port in the form of a message object. Once in the queue, the process waits for a processor for execution.

c. Dispatching

Dispatching is the assignment of processes to processors in the order in which the processes have been scheduled.

The meeting places for processors and processes are the Dispatcing Ports where the the processes stand in line waiting for service by a processor, and

processors go when they need a process to execute. Whenever a processor needs a process to execute, it simply removes the first process in line at its dispatching port and begins to execute it.

2. Dispatcing Ports and Program Structure

We have seen that the dispatching port objects are closely related to the processor object and process object which are part of the program structure. This relation must be well controlled in order to prevent perturbation of the program structure.

For example the network required to perform process scheduling and dispatching is complex. To keep track of these assignments two things must be accomplished. First, in each processor object there is a object reference for the processor's dispatching port, which means that each processor has only one dispatching port. This information forces a processor to always visit the same dispatching port. Second, in each process object there is an object reference to the dispatching port the process is allowed to visit and thus, a process can visit only one dispatching port. We can conclude, therefore, that by having one dispatcing port per processor and process the multiprocessing task of the 432 can be accomplished without major effort other than assigning the proper policy making parameters.

This chapter has presented the characteristics and advantages of the Intel 432 over conventinal microprocessors. Although this chapter is long and at times detailed, it presents the requisite information necessary to understand the results observed in the Benchmark Performance Chapter.

IV. MULTIPROCESSOR/MULTIPROCESS BENCHMARK PERFORMANCE

In general the Benchmark Performance Test was conducted using the Benchmark programs generated in Applegate [Ref. 3], modified as required to execute in the Multiprocessor/Multiprocess environment. Initially, the test was conducted with only one processor and one process in the system to confirm the results reported in Applegate [Ref. 3]. Then, using the same Benchmark Programs, the test was performed with two processor/processes, three processor/processes and finally four processor/processes in the system to determine if the 432/670 system is capable of sustaining incremental performance improvements as processors are added to the system.

A. BENCHMARK PERFORMANCE TEST

Kodres [Ref. 12] analyses multiprocessor systems that uses a common shared memory. Although dependent on the type of instructions being executed, it was shown that if the programs are fetched from common memory, the systems bus becomes a bottleneck with only two processors in the system. To determine if the INTEL 432/670 system could be effective with six processors as reported in the Intel manual [Ref. 13], a series of Benchmark performance tests were conducted using seven programs from the Computer Family Architecture Study (CFS) (Figures 4.1 - 4.4), two programs from a

performance comparison conducted by the University of California - Berkeley (Figure 4.5), and two programs that executed instructions at opposite ends of the iAPX 432 bus activity spectrum (Figure 4.6). All Benchmark programs were executed with the operating system configured for the default service period (DF) and again with the service period set to infinite period count (IF). Each figure depicting performance contains the name of the Benchmark program, the number of iterations, and the number of seconds required to complete the execution as well as the curve showing maximum theoretical efficiency, observed performance for the infinite period count (IF), and default (DF) service periods. Additionally, calculated efficiency numbers associated with each system configuration are included on the right side of the graph for easy reference.

Each program was executed with the INTEL 432/670 system configured for one processor. A second processor was installed and two processes containing identical Benchmark programs were timed during execution. Processors and processes were added incrementially up to and including four iAPX 432 General Data Processors (GDP). In general the tests show that the INTEL 432/670 system was still performing at approximately ninety percent efficiency with four processors in the system.

B. RESULTS

Since Kodres [Ref. 12] predicted serious bus contention at two processors and the observed performance is still at approximately ninety percent with four processors in the system, additional analysis was performed using a Logic State Analyzer to observe systems bus activity while executing instructions at opposite ends of the bus activity spectrum. Benchmark programs were designed and implemented that execute the MOVCH instruction to represent the high end of bus use density and DIVORD to represent the low end of bus activity.

Shoop [Ref. 14], considering a system bus width of sixteen bits and a high transfer speed, predicted that the MOVCH instruction would take 10.4 microseconds to execute while the DIVORD would require 29.6 microseconds. Using the Logic State Analyzer delay function and triggering on a known address, a series of program executions were performed to generate a bus activity snapshot of 650 microseconds. By counting the number of bus access cycles and the number of execution cycles, the relative efficiency of the Intel 432/670 computer system was calculated for one through 8 processors using the equation derived by Kodres [Ref. 12]: Figure 4.7 depicts the predicted and experimental results for the MOVCH and DIVORD instructions.

$$Re = (B + E) / (NB + I)$$ where Re = Relative efficiency

B = Bus activity cycles

E = Execution Cycles

I = System's Bus Idle Time

N = Number of Processors

To determine if the data gathered during the Bus Snapshot period was representative of the Benchmark performance executions, the instruction execution times are compared in Table 4.1. There is a small difference in execution times measured by the bus activity snapshot as opposed to timing the repeated execution of the instruction. Since the timing is accurate to the nearest second, this small difference can be attributed to measurement error. The larger difference between Shoop predicted execution times and measured execution times can be attributed to an overly optimistic prediction by Shoop.

From the calculations displayed in Figure 4.7, systems executing instructions consisting of MOVCH would begin to saturate the system's bus at 4.3 processors while DIVORD instructions induce the bottleneck at 7.7 processors. Since the execution time of DIVORD is greater than MOVCH, it follows that the predicted bus contention should be less for the DIVORD program as observed. Since these instructions represent bus activity at opposite ends of the bus activity

spectrum, it appears that the Intel's design of six processors per system is an appropriate choice.

C. ANALYSIS

The explanation for this performance improvement over that observed in other Multiprocessor/Multiprocess systems that fetch programs from common memory is the INTEL 432/670 system has a 32 bit system's bus. The 32 bit system's bus has a maximum of 77 megabits per second transfer rate compared to about 12 megabits per second for a typical 16 bit MULTIBUS system's bus. Additionally, the instruction formats used by the iAPX 432 processors are designed to reduce the number of bytes transferred. Finally, by using a byte addressable boundary rather than a word boundary, the number of bytes that must be transferred when a processor fetches instructions is reduced.

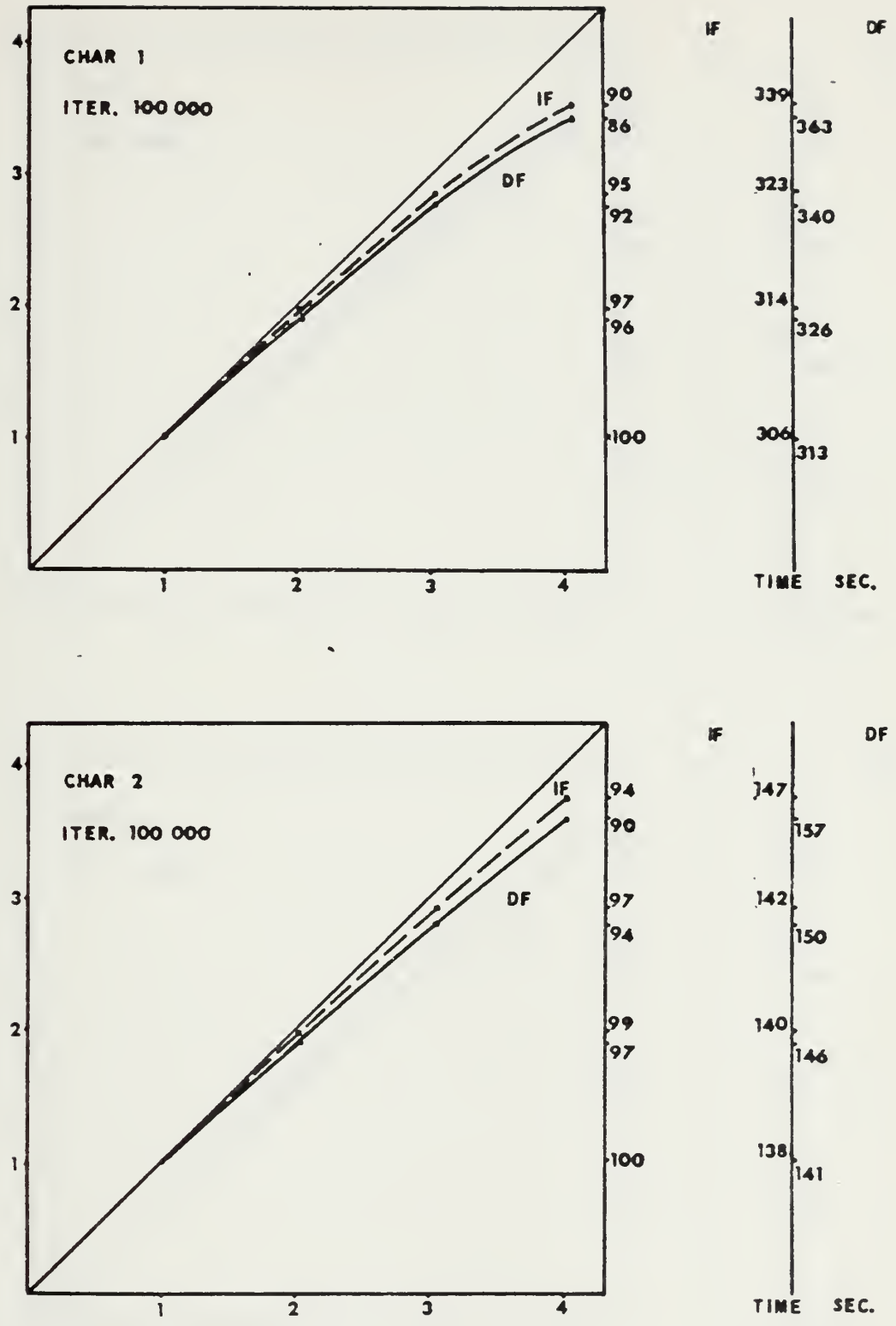


Figure 4.1 CFA Benchmark CHAR Performance.

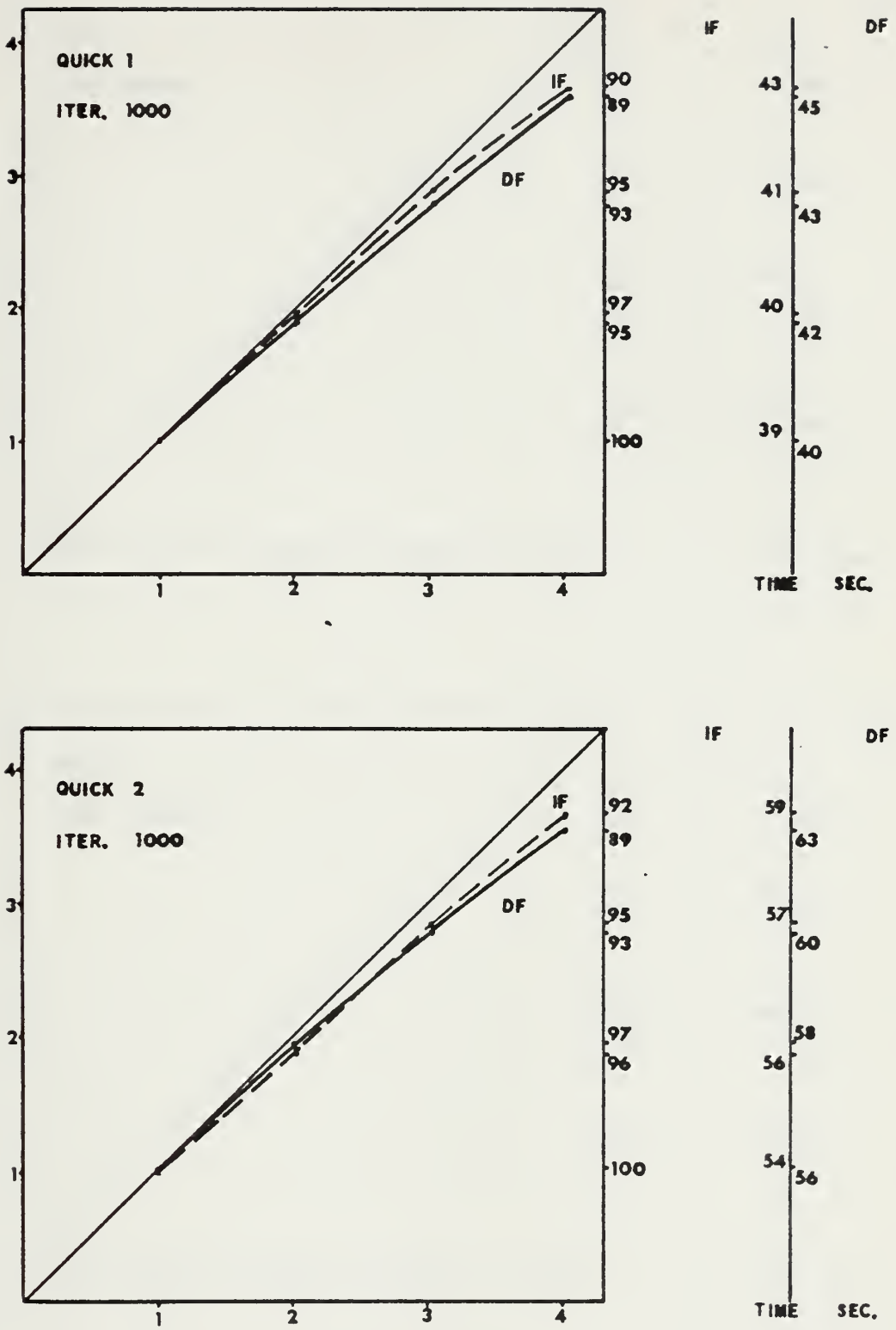


Figure 4.2 CFA Benchmark QUICK Performance.

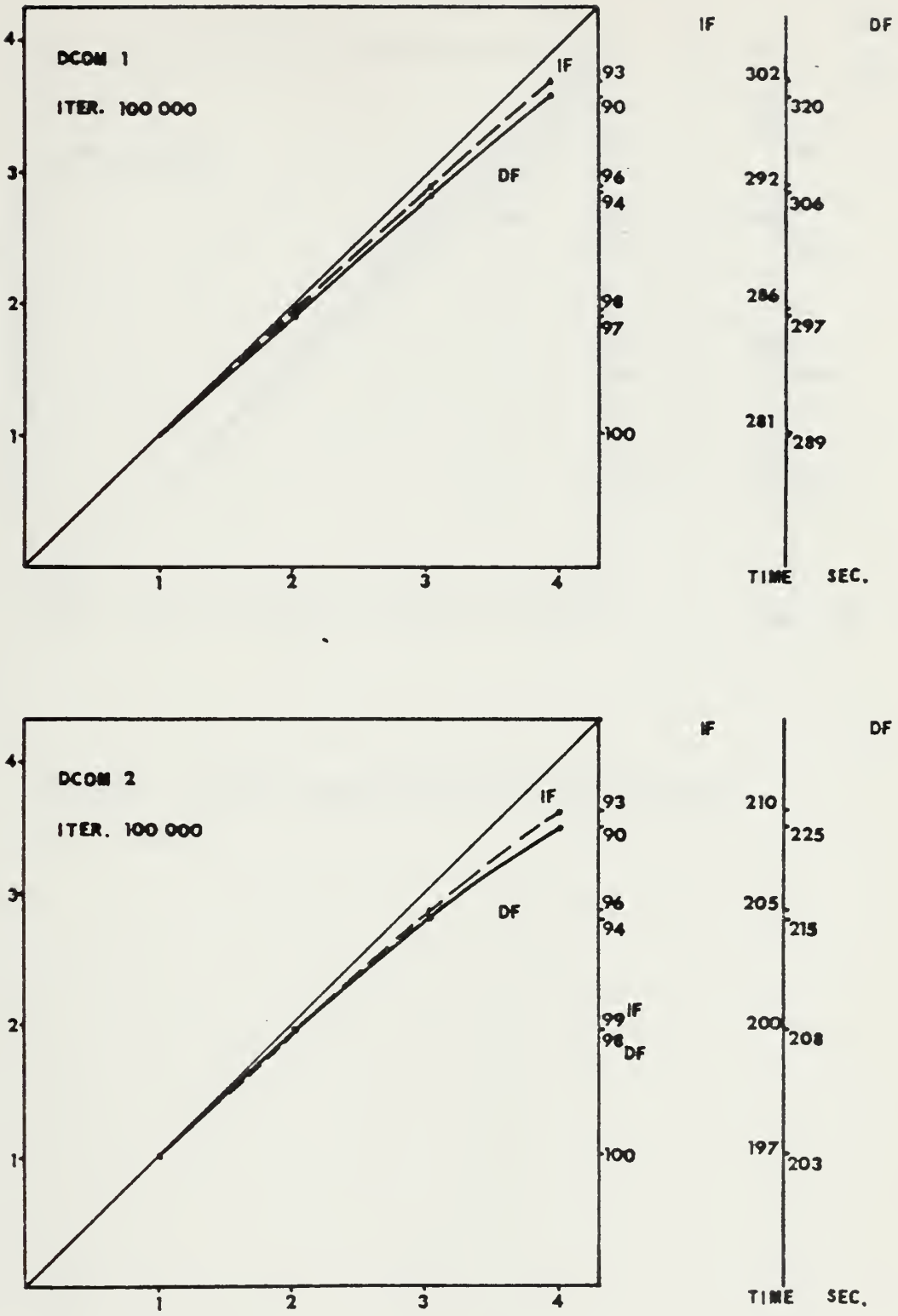


Figure 4.3 CFA Benchmark DECOM Performance.

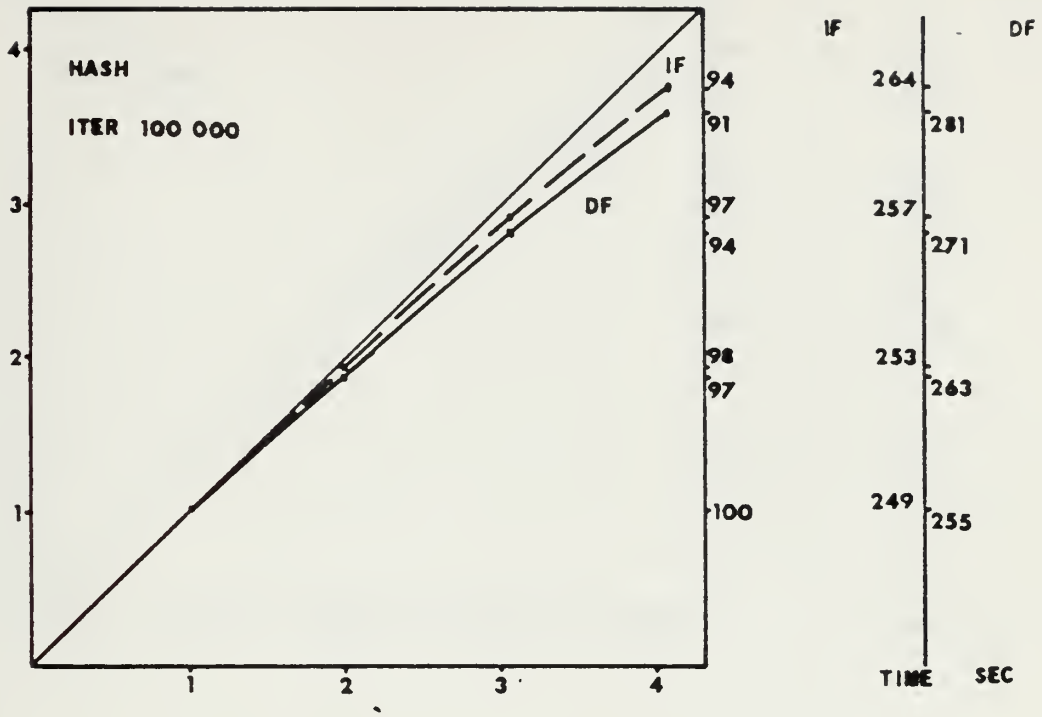


Figure 4.4 CFA Benchmark HASH Performance.

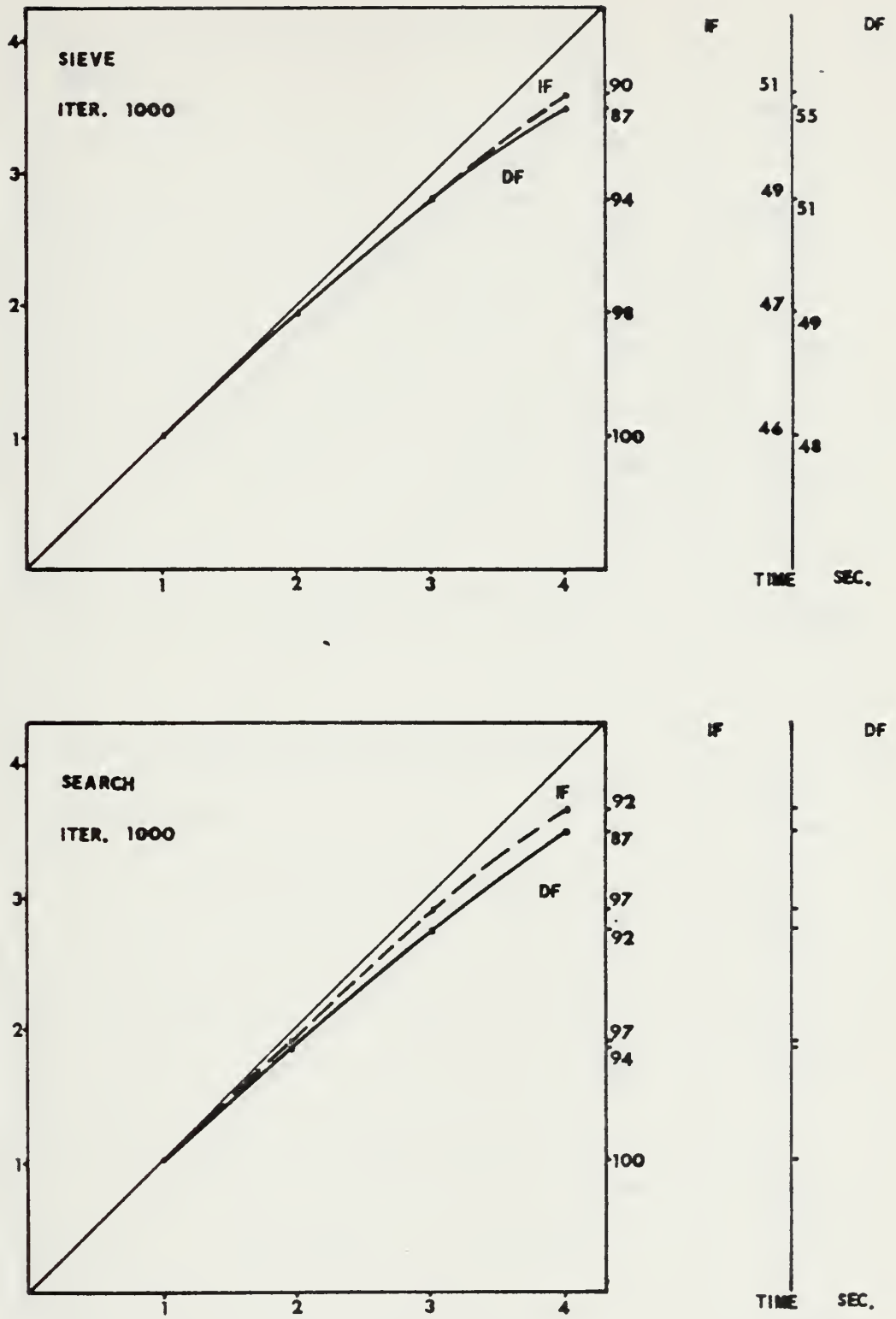
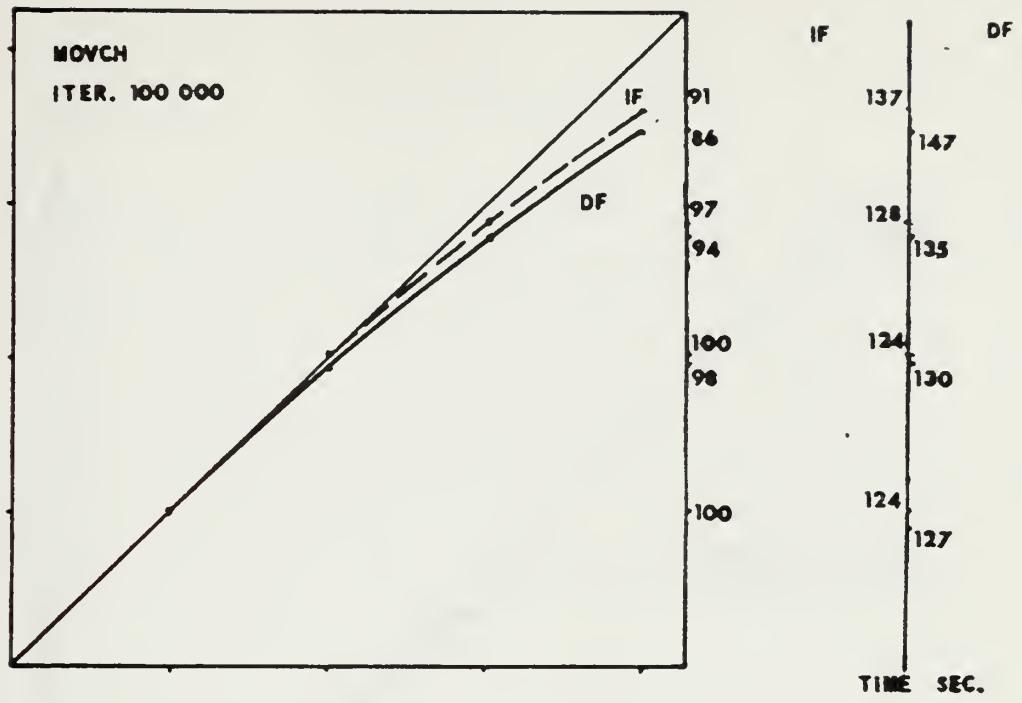


Figure 4.5 U. C. Berkeley Benchmark Performance.



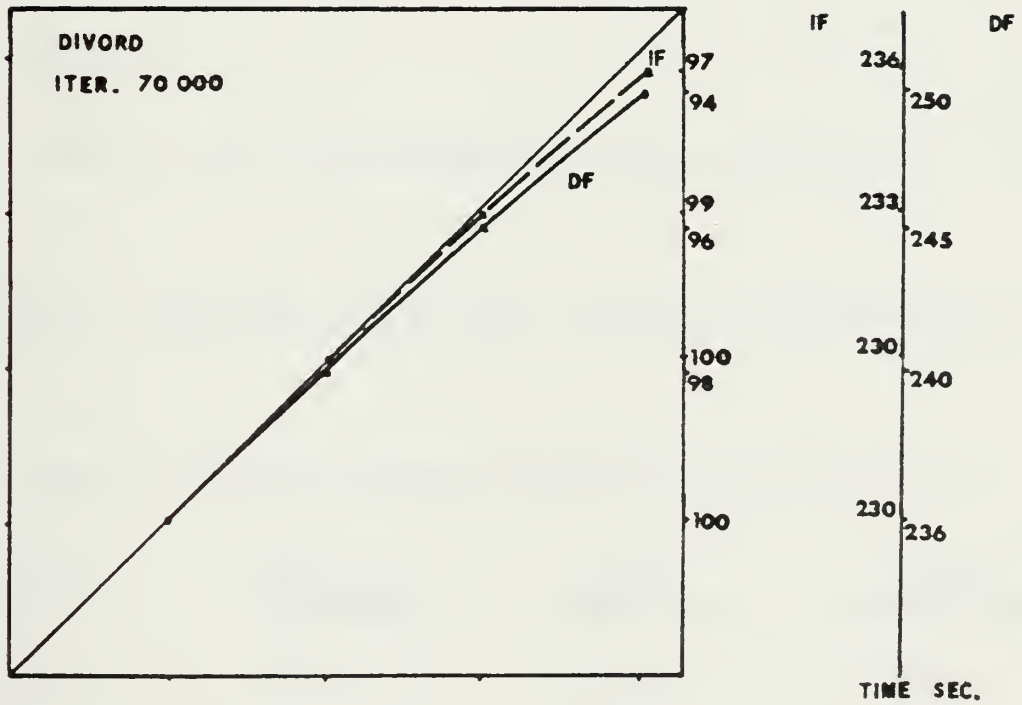


Figure 4.6 MAX/MIN Bus Activity Benchmark Performance.

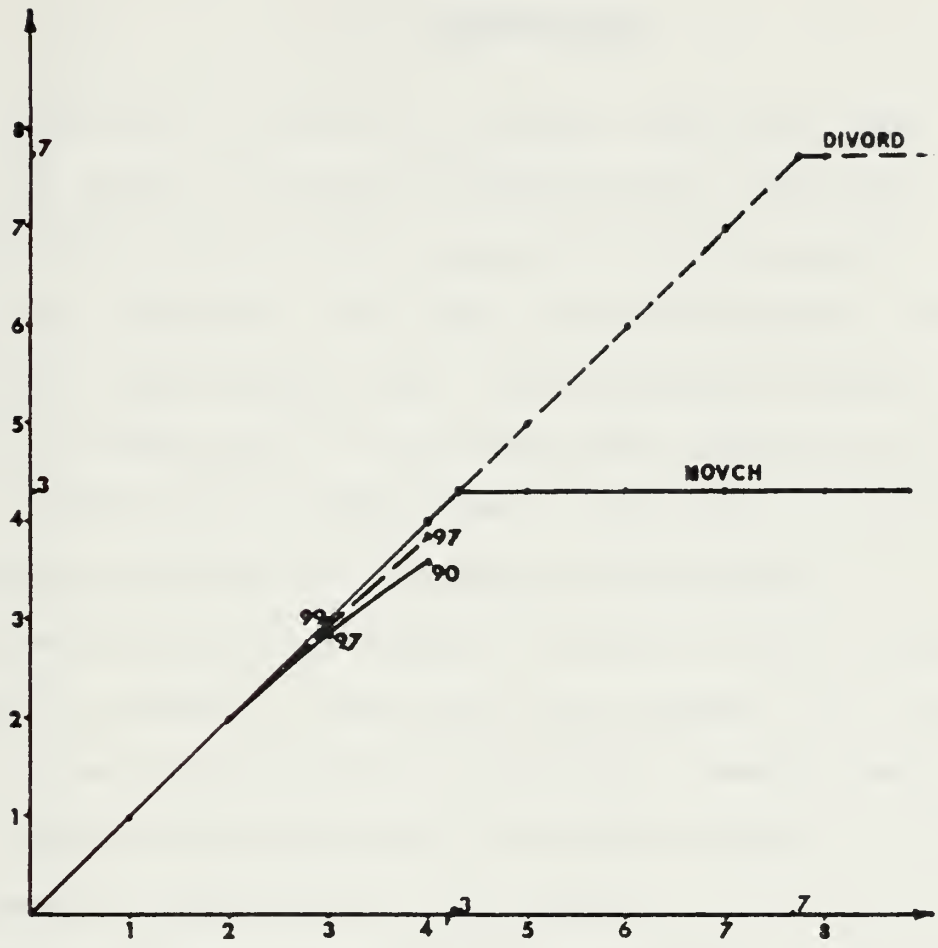


Figure 4.7 Predected and Actual Processing Performance.

Table 4.1 Instruction Execution Times (Microseconds).

INSTRUCTION	432/570 OBSERVED	BUS SNAPSHOT	SHOOP PREDICTED
MOVCH	12.4	12.6	10.4
DIVORD	32.3	32.1	29.6

V. CONCLUSIONS

The excessive execution overhead resulting from the Intel 432 object oriented architecture has prevented the Intel 432 from demonstrating superiority when comparing its speed of execution to other microprocessors in a Uniprocessor environment (ie., one processor with its own memory and systems bus). For example when compared with the Motorola 68000 and the INTEL 8086, the iAPX 432 provided approximately one-half the throughput [PATTERSON].

Benchmark performance depends heavily on the instruction set being executed. Since individual systems use unique methods of performing certain operations, average Benchmark figures could be misleading if the system was to be designed for a specific application. For example, if the instruction set required was predominately Floating Point operations, the INTEL 432 performance should be superior because of its efficient Floating Point hardware.

When comparing a computer's ability to perform in a Multiprocessor environment, instruction mix is still important but you must also consider the bandwidth of the memory. There are three types of Multiprocessor architectures that effect the bandwidth characteristics of Multiprocessor systems:

1. Program and data stored in global memory where all processors have equal access.

2. Program stored in a local memory isolated to access only by the parent processor. Data would be stored in global memory.

3. Same as system 2 except local memories are also accessible by other processors through a Crossbar Switch.

The Crossbar Switch is a system that enables the designer to increase the number of system buses to the point where there is a separate path available for each memory unit. Its obvious advantage is to eliminate the system bus contention when operating in a Multiprocessor/Multiprocess environment. Since many real-time applications programs reside in global memory, the Benchmark test were conducted in the global memory stored program environment. Processing power of a system of computers sharing a common systems bus, is measured in relative efficiency (see Chapter IV. B.). Each processor must be able to process the program steps and data without interfering with each other by saturating the system bus with request for instruction and data fetches.

Analysis of the INTEL 432/670 system Benchmark performance test show that the relative efficiency of the system is still approximately 90 percent with four processors in the system. The effectively maximum number of processors is confirmed at six. This is in sharp contrast to an effectively maximum of two processors for the INTEL 8086 operating with a MULTIBUS systems bus [KODRES]. Again it is

empnasized tnat tnese performance figures are based on a common memory stored program system.

Although tnis tnesis was not concerned with transparent multiprocessing, tne advantages of a system tnat supports this concept was evident. During Benchmark program executions, a processor was prevented from being initialized due to a loose piece of plastic in tne motherboard card edge connector. The system continued to function automatically sequencing tne extra process into tne dispatching queue. The actual failure was only discovered when attempting to determine a sudden increase in execution times.

Benchmark programs for this project were written in the Department of Defense programming language Ada. Previously generated code was found to be self-documenting, easy to modify, and interfacing problems were non-existant. Although Ada has been cited as a complex language, tne ease with wnich it is modified combined with its strong typing characteristics, and readability confirms its ability to support tne Software Engineering principles.

While tne effectively maximum number of processors for general purpose computing in the INTEL 432/670 system is comfirmed at six processors, incorporating tne iAPX 432 into a Crossbar Switch system could produce a system capable of supporting up to 50 processors. Its direct military application to tactical systems requiring a growth potential

is exciting particularly when considering the ability of the iAPX 432 to support transparent multiprocessing.

Since current Navy microcomputers such as the AN/AYK-7 cannot be expanded beyond four processors without requiring major restructuring of the existing software, it is recommended that a follow on study be conducted to determine if a large system is feasible using the iAPX 432 in a Crossbar Switch system.

APPENDIX A

iAPX 432 VS CONVENTIONAL MAINFRAMES

This appendix provides a comparison of the iAPX 432 Architecture and the Architecture of Conventional Mainframes.

Feature of Archit.	Convent. Mainfr Architecture	iAPX 432 Architecture
MEMORY ORGANIZATION		
Organization, size	Linear 2**24-2**32 bytes	Structured segmented 2**24 segments 2**16 byte displacement 2**40 byte virtual address space
Logical to physical address translation	Single-level map Page-based reloc and virtual memor	Two-level map Segment-based reloc and virtual memory
Protected memory unit	Fixed-size page	Individual program module or data structure
DATA MANIPULATION		
Expression evaluation	General register	Stack or memory-to-memory
Primitive data types	Characters, unsigned integers integers, reals	Characters, unsigned integers, integers, reals, temporary reals
Floating point hardware	Yes	Yes
Addressing modes	Some modes not available for all operands	Symmetrical: all modes available for every operand
PROGRAMMING ENVIRONMENT SUPPORT		
Operating system	No multi-process support No support for dynamic storage alloctation	Multi-process mechanisms in hardware Dynamic storage allocation mechanisms in hardware

	Very limited mult multi- processor operation if any	Software-transparent multi-processor operation
Hign level language	Assembly language oriented instruction set	Oriented toward nigh level languages
Programming methodology	No support at all	Object-based architecture

APPENDIX B

PIN DESCRIPTION

This appendix provides the pin description of the iAPX 432 general data processor.

43201 Pin Description.

Processor Packet Bus Group.

ACD15-ACDØ (Address/Control/Data lines, Inputs, high asserted).
PRQ (Processor Packet bus Request,Input,high asserted).
ICS (Interconnect Status, Input, high asserted).
Intra-GDP Bus Group.
UI15--UIØ (Microinstruction Bus lines, Outputs, high asserted).
IS6---ISØ (Interchip Status lines, Inputs, high asserted).

System Group

FATAL/(Fatal, Output,low asserted).
ALARM/(Alarm signal,Input,low asserted).
INIT/(Initialization,Input,low asserted).
CLR/(Clear,Input,low asserted)
Hardware Error Detection Group
Master (Master,Input,high asserted).
HERR/(Hardware Error,Output,low asserted).

Clock Group

CLKA, CLKB (Clock A, Clock B, Inputs).

Testing Input

RDROM/(Read ROM,Input,low asserted)

Power and Ground Connections.

VCC (4 pins).
GND (5 pins)
VBB (Internally Generated).
N.C. (No Connection, 4 pins)

43202 Pin Description

Processor Packet Bus Group

PRQ (Processor Packet request, Three-state Output, high asserted).
ICS (Interconnect Status, Input, high asserted).
Bout (Enable Buffers for Output, Output,high asserted).

Intra-GDP Bus Group.

uI15-uI0 (Microinstruction Bus lines, Inputs, high asserted).
IS6-IS0(Interchip Status lines, Outputs, high asserted).

System Group

PCLK/(Processor Clock, Input, low asserted).
CLR/(Clear,Input,low asserted).

Hardware Error Detection Group.

Master (Master, Input, high assered; 25K nomonal pullup on-chip).
HERR/(Hardware Error, Open Drain Output, low asserted).

Clock Group

CLKA, CLKB (Clock A, Clock B, Inputs).

Power and Ground Connections.

Vcc (Power Supply, 4 pins).
GND (Ground, 5 pins).
N.C. (No Connection, 7 pins).

APPENDIX C

SAMPLE PROGRAM LISTING WITH SUBMIT FILES

This appendix contains all the necessary files to generate Multiprocessor/Multiprocesses environment.

CHARS1.MSS

```
-- CHARS1  package specification
--
-- This is the ADA specification package for the
-- CFA character search benchmark.
--
-- CHARS1
--
package SCHAR is
 subtype subint is integer range 1..256;
 type txtarray is array(1..256) of character;
 array1,array2 : txtarray;
 procedure SEARCH(srchlen,arglen : IN subint;
                  array1,array2  : IN txtarray;
                             loc : OUT integer);
end SCHAR;
```

CHARS1.MBS

```
-- CHARS1    package body
--
--
-- The following package contains the procedures needed to
-- implement the CFA character search benchmark. Procedure
--  SEARCH performs the actual search as desired by the CFA.
--
-- CHARS1
--
--
pragma environment("ACS:TEXTIO.MLE","INTIO.MSE",
                                              "SCHAR.MSE");
with text_io,intio; use text_io,intio,ascii;
package body SCHAR is
 procedure SEARCH(srchlen,arglen : IN subint;
                  array1,array2  : IN txtarray;
                             loc : OUT integer) is
```

```
  i,j : integer;
  begin
   i := 1;
   j := 1;
   loc := -1;
   while i <= srcnlen loop
    if array1(i) = array2(j) then
     if j+1 <= arglen then
      i := i+1;
      j := j+1;
     else
      loc := i-j;
      exit;
     end if;
    else
     i := i+1;
     j := 1;
    end if;
   end loop;
 end SEARCH;
end SCHAR;
```

PR1MAIN.MSS

```
package USER_PROCESS_1 is
 procedure MAIN;
end USER_PROCESS_1;
```

PR2MAIN.MSS

```
package USER_PROCESS_2 is
 procedure MAIN;
end USER_PROCESS_2;
```

PR3MAIN.MSS

```
package USER_PROCESS_3 is
 procedure MAIN;
end USER_PROCESS_3;
```

PR4MAIN.MSS

```
package USER_PROCESS_4 is
 procedure MAIN;
end USER_PROCESS_4;
```

PR1MAIN.MBS

```
--CHARS1  driver routine
--
--
--
pragma environment("ACS:TEXTIO.MLE","INTIO.MSE","SCHAR.MSE",
                   "PR1MAIN.MSE");
with text_io,intio,schar; use text_io,intio,schar,ascii;
--
--
--
--          Timing also includes time for procedure
--          invocation
--          3 May 1983
--
package body USER_PROCESS_1 is
 procedure MAIN is
  i,loc,srcn_lengtn,srcn_arg,timer_loop : integer;
--
   begin
  --
  --
  -- initialization
  --
     array1 := ('M','o','n','d','a','y',',',' ','J','u','n',
                'e','7','t','h',',',' ','1','9','7','6',
                                          others => ' ');
     array2 := ('d','a','y',others => ' ');
     srcn_length := 22;
     srcn_arg    := 3;
     timer_loop  := 100000;
       put_20(" Start of Search...1");
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
       for i in 1.. timer_loop loop
       SEARCH(srcn_lengtn,srcn_arg,array1,array2,loc);
       end loop;
       put_20("end the search.....1");
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
 end MAIN;
end USER_PROCESS_1;
```

PR2MAIN.MBS

```
--CHARS1  driver routine
--
--
--
pragma environment("ACS:TEXTIO.MLE","INTIO.MSE","SCHAR.MSE",
                   "PR2MAIN.MSE");
with text_io,intio,schar; use text_io,intio,scnar,ascii;
--
--
--
--          Timing also includes time for procedure
--          invocation.
--          3 may 1983
--
package body USER_PROCESS_2 is
 procedure MAIN is
  i,loc,srch_length,srcn_arg,timer_loop:integer;
--
   begin
  --
  --
  -- initialization
  --
     array1 := ('M','o','n','d','a','y',',',' ','J','u','n',
                'e','7','t','n',',',' ','1','9','7','6',
                                          others => ' ');
     array2 := ('d','a','y',others => ' ');
     srch_length := 22;
     srcn_arg    := 3;
     timer_loop  := 100000;
       put_20(" Start of Searcn...2");
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
       for i in 1.. timer_loop loop
       SEARCH(srcn_lengtn,srcn_arg,array1,array2,loc);
       end loop;
       put_20("end tne searcn.....2");
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
 end MAIN;
end USER_PROCESS_2;
```

PR3MAIN.MBS

```
--CHARS1  driver routine
--
--
--
pragma environment("ACS:TEXTIO.MLE","INTIO.MSE","SCHAR.MSE",
                   "PR3MAIN.MSE");
with text_io,intio,scnar; use text_io,intio,scnar,ascii;
--
--
--
--          Timing also includes time for procedure
--          invocation.
--          3 May 1983
--
package body USER_PROCESS_3 is
 procedure MAIN is
  i,loc,srch_length,srch_arg,timer_loop : integer;
--
   begin
  --
  --
  -- initialization
  --
     array1 := ('M','o','n','d','a','y',',',' ','J','u','n',
                'e','7','t','n',',',' ','1','9','7','5',
                                                 others => ' ');
     array2 := ('d','a','y',others => ' ');
     srch_length := 22;
     srch_arg    := 3;
     timer_loop  := 100000;
       put_20(" Start of Search...3");
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
       for i in 1.. timer_loop loop
       SEARCH(srch_length,srch_arg,array1,array2,loc);
       end loop;
       put_20("end the search.....3");
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
 end MAIN;
end USER_PROCESS_3;
```

PR4MAIN.MBS

```
--CHARS1  driver routine
--
--
--
pragma environment("ACS:TEXTIO.MLE","INTIO.MSE","SCHAR.MSE",
                   "PR4MAIN.MSE");
with text_io,intio,schar; use text_io,intio,schar,ascii;
--
--
--          Timing also includes time for procedure
--          invocation.
--          3 May 1983
--
package body USER_PROCESS_4 is
 procedure MAIN is
  i,loc,srch_length,srch_arg,timer_loop : integer;
  forever : boolean :=true;
  answer : character;
--
   begin
  --
    while forever loop
  --
  -- initialization
  --
     array1 := ('M','o','n','d','a','y',',',' ','J','u','n',
                'e','7','t','h',',',' ','1','9','7','6',
                                          others => ' ');
     array2 := ('d','a','y',others => ' ');
     srch_length := 22;
     srch_arg    := 3;
     timer_loop  := 100000;
       put_20(" Start of Search...4");
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
       for i in 1.. timer_loop loop
       SEARCH(srch_length,srch_arg,array1,array2,loc);
       end loop;
       put_20("end the search.....4");
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
       put(BEL);
        forever := false;
  end loop;
 end MAIN;
end USER_PROCESS_4;
```

DFPSERP1.MBS

```
Pragma environment("ACS:BPM.MLE","ACS:V1USRP.MLI",
   "PR1MAIN.MSE","PR2MAIN.MSE","PR3MAIN.MSE","PR4MAIN.MSE");
with Process_Definitions, iMAX_Definitions,
                                  Basic_Process_management;
with User_Process_1,User_Process_2,User_Process_3,
                                              User_Process_4;
package body User_Processes is

  -- Function:
  --   This module instantiates the user processes and
  --   provides an initialize procedure which completes the
  --   initialization of each user process.
  --
  --   For each user process, there is an instance of
  --   iMAX_Definitions.procedure_val which contains the
  --   process' initial procedure.  To minimize the size of
  --   this package, the initial procedure here simply calls
  --   the actual initial procedure which is found in
  --   another module.  For each process, there is also an
  --   instantiation of Process_Definitions.Process_Instance
  --   which specifies the process id, the procedure_val
  --   just mentioned, and the process' stack SRO and object
  --   table sizes.
  --
  --   Finally, the initialize procedure provided by this
  --   module simply calls the
  --   Complete_process_initialization procedure
  --   in each instantiation of
  --   Process_Definitions.Process_Instance.

  procedure pr1main;
  procedure pr2main;
  procedure pr3main;
  procedure pr4main;

  ------------------------------------
  -- SPECIFICATIONS FOR USER PROCESS 1 --
  ------------------------------------

  package UPROC1 is new Process_Definitions.Process_Instance
    (Process_Startup => pr1main,
     SRO_size => 4000);
-- service_period =>
            basic_process_management.infinite_period_count);
```

```
-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 2 --
-------------------------------------

package UPROC2 is new Process_Definitions.Process_Instance
  (Process_Startup => pr2main,
   SRO_size => 4000);
-- service_period =>
            basic_process_management.infinite_period_count);

-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 3 --
-------------------------------------

package UPROC3 is new Process_Definitions.Process_Instance
  (Process_Startup => pr3main,
   SRO_size => 4000);
-- service_period =>
            basic_process_management.infinite_period_count);

-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 4 --
-------------------------------------

package UPROC4 is new Process_Definitions.Process_Instance
  (Process_Startup => pr4main,
   SRO_size => 4000);
-- service_period =>
            Basic_Process_Management.Infinite_Period_count);

  -------------------------------
  -- BODIES FOR USER PROCESS 1 --
  -------------------------------

  procedure pr1main is
  begin
    User_Process_1.MAIN;
  end pr1main;

-------------------------------
-- BODIES FOR USER PROCESS 2 --
-------------------------------

procedure pr2main is
begin
  User_Process_2.main;
end pr2main;
```

```
------------------------------
-- BODIES FOR USER PROCESS 3 --
------------------------------

procedure pr3main is
begin
  User_Process_3.main;
end pr3main;


------------------------------
-- BODIES FOR USER PROCESS 4 --
------------------------------

procedure pr4main is
begin
  User_Process_4.main;
end pr4main;

  ------------------------------------------------
  -- PROCEDURE TO INITIALIZE ALL USER PROCESSES --
  ------------------------------------------------
  procedure Initialize
  is
  begin
    UPROC1.Complete_process_initialization;
--  UPROC2.Complete_process_initialization;
--  UPROC3.Complete_process_initialization;
--  UPROC4.Complete_process_initialization;
  end Initialize;
end User_Processes;
```

DFPSERP2.MBS

```
Pragma environment("ACS:BPM.MLE","ACS:V1USRP.MLI",
   "PR1MAIN.MSE","PR2MAIN.MSE","PR3MAIN.MSE","PR4MAIN.MSE");
with Process_Definitions, iMAX_Definitions,
                                   Basic_Process_management;
with User_Process_1,User_Process_2,User_Process_3,
                                                User_Process_4;
package body User_Processes is

  -- Function:
  --   This module instantiates the user processes and
  --   provides an initialize procedure which completes the
  --   initialization of each user process.
  --
  --   For each user process, there is an instance of
  --   iMAX_Definitions.procedure_val which contains the
  --   process' initial procedure.  To minimize the size of
  --   this package, the initial procedure here simply calls
  --   the actual initial procedure which is found in
  --   another module.  For each process, there is also an
  --   instantiation of Process_Definitions.Process_Instance
  --   which specifies the process id, the procedure_val
  --   just mentioned, and the process' stack SRO and object
  --   table sizes.
  --
  --   Finally, the initialize procedure provided by this
  --   module simply calls the
  --   Complete_process_initialization procedure in each
  --   instantiation of Process_Definitions.Process_Instance

  procedure pr1main;
  procedure pr2main;
  procedure pr3main;
  procedure pr4main;

  ---------------------------------------
  -- SPECIFICATIONS FOR USER PROCESS 1 --
  ---------------------------------------

  package UPROC1 is new Process_Definitions.Process_Instance
    (Process_Startup => pr1main,
     SRO_size => 4000);
-- service_period =>
           basic_process_management.infinite_period_count);
```

```
-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 2 --
-------------------------------------

package UPROC2 is new Process_Definitions.Process_Instance
  (Process_Startup => pr2main,
   SRO_size => 4000);
-- service_period =>
            basic_process_management.infinite_period_count);


-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 3 --
-------------------------------------

package UPROC3 is new Process_Definitions.Process_Instance
  (Process_Startup => pr3main,
   SRO_size => 4000);
-- service_period =>
            basic_process_management.infinite_period_count);

-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 4 --
-------------------------------------

package UPROC4 is new Process_Definitions.Process_Instance
  (Process_Startup => pr4main,
   SRO_size => 4000);
-- service_period =>
            Basic_Process_Management.Infinite_Period_count);

  -------------------------------
  -- BODIES FOR USER PROCESS 1 --
  -------------------------------

  procedure pr1main is
  begin
    User_Process_1.MAIN;
  end pr1main;

-------------------------------
-- BODIES FOR USER PROCESS 2 --
-------------------------------

procedure pr2main is
begin
  User_Process_2.main;
end pr2main;
```

```
------------------------------
-- BODIES FOR USER PROCESS 3 --
------------------------------

procedure pr3main is
begin
  User_Process_3.main;
end pr3main;

------------------------------
-- BODIES FOR USER PROCESS 4 --
------------------------------

procedure pr4main is
begin
  User_Process_4.main;
end pr4main;

  ------------------------------------------------
  -- PROCEDURE TO INITIALIZE ALL USER PROCESSES --
  ------------------------------------------------
  procedure Initialize
  is
  begin
    UPROC1.Complete_process_initialization;
    UPROC2.Complete_process_initialization;
--  UPROC3.Complete_process_initialization;
--  UPROC4.Complete_process_initialization;
  end Initialize;

end User_Processes;
```

DFPSERP3.MBS

```
Pragma environment("ACS:BPM.MLE","ACS:V1USRP.MLI",
   "PR1MAIN.MSE","PR2MAIN.MSE","PR3MAIN.MSE","PR4MAIN.MSE");
with Process_Definitions, iMAX_Definitions,
                                  Basic_Process_management;
with User_Process_1,User_Process_2,User_Process_3,
                                              User_Process_4;
package body User_Processes is

  -- Function:
  --   This module instantiates the user processes and
  --   provides an initialize procedure which completes the
  --   initialization of each user process.
  --
  --   For each user process, there is an instance of
  --   iMAX_Definitions.procedure_val which contains the
  --   process' initial procedure.  To minimize the size of
  --   this package, the initial procedure here simply calls
  --   the actual initial procedure which is found in
  --   another module.  For each process, there is also an
  --   instantiation of Process_Definitions.Process_Instance
  --   which specifies the process id, the procedure_val
  --   just mentioned, and the process' stack SRO and object
  --   table sizes.
  --
  --   Finally, the initialize procedure provided by this
  --   module simply calls the
  --   Complete_process_initialization procedure in each
  --   instantiation of Process_Definitions.Process_Instance

  procedure pr1main;
  procedure pr2main;
  procedure pr3main;
  procedure pr4main;

  ---------------------------------------
  -- SPECIFICATIONS FOR USER PROCESS 1 --
  ---------------------------------------

  package UPROC1 is new Process_Definitions.Process_Instance
    (Process_Startup => pr1main,
     SRO_size => 4000);
-- service_period =>
            basic_process_management.infinite_period_count);
```

```
--------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 2 --
--------------------------------------

package UPROC2 is new Process_Definitions.Process_Instance
  (Process_Startup => pr2main,
   SRO_size => 4000);
-- service_period =>
            basic_process_management.infinite_period_count);

--------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 3 --
--------------------------------------

package UPROC3 is new Process_Definitions.Process_Instance
  (Process_Startup => pr3main,
   SRO_size => 4000);
-- service_period =>
            basic_process_management.infinite_period_count);

--------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 4 --
--------------------------------------

package UPROC4 is new Process_Definitions.Process_Instance
  (Process_Startup => pr4main,
   SRO_size => 4000);
-- service_period =>
            Basic_Process_Management.Infinite_Period_count);

  ------------------------------
  -- BODIES FOR USER PROCESS 1 --
  ------------------------------

  procedure pr1main is
  begin
    User_Process_1.MAIN;
  end pr1main;


------------------------------
-- BODIES FOR USER PROCESS 2 --
------------------------------

procedure pr2main is
begin
  User_Process_2.main;
end pr2main;
```

```
------------------------------
-- BODIES FOR USER PROCESS 3 --
------------------------------

procedure pr3main is
begin
  User_Process_3.main;
end pr3main;

------------------------------
-- BODIES FOR USER PROCESS 4 --
------------------------------

procedure pr4main is
begin
  User_Process_4.main;
end pr4main;

  ------------------------------------------------
  -- PROCEDURE TO INITIALIZE ALL USER PROCESSES --
  ------------------------------------------------
  procedure Initialize
  is
  begin
    UPROC1.Complete_process_initialization;
    UPROC2.Complete_process_initialization;
    UPROC3.Complete_process_initialization;
--  UPROC4.Complete_process_initialization;
  end Initialize;

end User_Processes;
```

DFPSERP4.MBS

```
Pragma environment("ACS:BPM.MLE","ACS:V1USRP.MLI",
    "PR1MAIN.MSE","PR2MAIN.MSE","PR3MAIN.MSE","PR4MAIN.MSE");
with Process_Definitions, iMAX_Definitions,
                                    Basic_Process_management;
with User_Process_1,User_Process_2,User_Process_3,
                                          User_Process_4;
package body User_Processes is

  -- Function:
  --   This module instantiates the user processes and
  --   provides an initialize procedure which completes the
  --   initialization of each user process.
  --
  --   For each user process, there is an instance of
  --   iMAX_Definitions.procedure_val which contains the
  --   process' initial procedure.  To minimize the size of
  --   this package, the initial procedure here simply calls
  --   the actual initial procedure which is found in
  --   another module.  For each process, there is also an
  --   instantiation of Process_Definitions.Process_Instance
  --   which specifies the process id, the procedure_val
  --   just mentioned, and the process' stack SRO and object
  --   table sizes.
  --
  --   Finally, the initialize procedure provided by this
  --   module simply calls the
  --   Complete_process_initialization procedure in each
  --   instantiation of Process_Definitions.Process_Instance

  procedure pr1main;
  procedure pr2main;
  procedure pr3main;
  procedure pr4main;

  ---------------------------------------
  -- SPECIFICATIONS FOR USER PROCESS 1 --
  ---------------------------------------

  package UPROC1 is new Process_Definitions.Process_Instance
    (Process_Startup => pr1main,
     SRO_size => 4000);
-- service_period =>
            basic_process_management.infinite_period_count);
```

```
--------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 2 --
--------------------------------------

package UPROC2 is new Process_Definitions.Process_Instance
  (Process_Startup => pr2main,
   SRO_size => 4000);
-- service_period =>
            basic_process_management.infinite_period_count);

--------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 3 --
--------------------------------------

package UPROC3 is new Process_Definitions.Process_Instance
  (Process_Startup => pr3main,
   SRO_size => 4000);
-- service_period =>
            basic_process_management.infinite_period_count);
```

```
-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 4 --
-------------------------------------

package UPROC4 is new Process_Definitions.Process_Instance
  (Process_Startup => pr4main,
   SRO_size => 4000);
-- service_period =>
            Basic_Process_Management.Infinite_Period_count);

  -------------------------------
  -- BODIES FOR USER PROCESS 1 --
  -------------------------------

  procedure pr1main is
  begin
    User_Process_1.MAIN;
  end pr1main;

-------------------------------
-- BODIES FOR USER PROCESS 2 --
-------------------------------

procedure pr2main is
begin
  User_Process_2.main;
end pr2main;

-------------------------------
-- BODIES FOR USER PROCESS 3 --
-------------------------------

procedure pr3main is
begin
  User_Process_3.main;
end pr3main;

-------------------------------
-- BODIES FOR USER PROCESS 4 --
-------------------------------

procedure pr4main is
begin
  User_Process_4.main;
end pr4main;
```

```
  ------------------------------------------------
  -- PROCEDURE TO INITIALIZE ALL USER PROCESSES --
  ------------------------------------------------
  procedure Initialize
  is
  begin
    UPROC1.Complete_process_initialization;
    UPROC2.Complete_process_initialization;
    UPROC3.Complete_process_initialization;
    UPROC4.Complete_process_initialization;
  end Initialize;

end User_Processes;
```

DFLINK1.LKD

```
; instruction test program

link ACS:minmax.eod
ACS:textio.mlo
intio.mso
schar.mso
schar.mbo
pr1main.mso
pr2main.mso
pr3main.mso
pr4main.mso
pr1main.mbo
pr2main.mbo
pr3main.mbo
pr4main.mbo
dfpserp1.mbo
psors.mbo

free(30 in directory)

output dfchar1.eod

objectmap
```

DFLINK2.LKD

```
; instruction test program

link ACS:minmax.eod
ACS:textio.mlo
intio.mso
schar.mso
schar.mbo
pr1main.mso
pr2main.mso
pr3main.mso
pr4main.mso
pr1main.mbo
pr2main.mbo
pr3main.mbo
pr4main.mbo
dfpserp2.mbo
psors.mbo

free(30 in directory)

output dfchar2.eod

objectmap
```

DFLINK3.LKD

```
; instruction test program

link ACS:minmax.eod
ACS:textio.mlo
intio.mso
schar.mso
scnar.mbo
pr1main.mso
pr2main.mso
pr3main.mso
pr4main.mso
pr1main.mbo
pr2main.mbo
pr3main.mbo
pr4main.mbo
dfpserp3.mbo
psors.mbo

free(30 in directory)

output dfcnar3.eod

objectmap
```

DFLINK4.LKD

```
; instruction test program

link ACS:minmax.eod
ACS:textio.mlo
intio.mso
schar.mso
scnar.mbo
pr1main.mso
pr2main.mso
pr3main.mso
pr4main.mso
pr1main.mbo
pr2main.mbo
pr3main.mbo
pr4main.mbo
ifpserp4.mbo
psors.mbo

free(30 in directory)

output ifcnar4.eod

objectmap
```

INFPSERP1.MBS

```
Pragma environment("ACS:BPM.MLE","ACS:V1USRP.MLI",
   "PR1MAIN.MSE","PR2MAIN.MSE","PR3MAIN.MSE","PR4MAIN.MSE");
with Process_Definitions, iMAX_Definitions,
                                   Basic_Process_management;
with User_Process_1,User_Process_2,User_Process_3,
                                              User_Process_4;
package body User_Processes is

  -- Function:
  --   This module instantiates the user processes and
  --   provides an initialize procedure which completes the
  --   initialization of each user process.
  --
  --   For each user process, there is an instance of
  --   iMAX_Definitions.procedure_val which contains the
  --   process' initial procedure.  To minimize the size of
  --   this package, the initial procedure here simply calls
  --   the actual initial procedure which is found in
  --   another module.  For each process, there is also an
  --   instantiation of Process_Definitions.Process_Instance
  --   which specifies the process id, the procedure_val
  --   just mentioned, and the process' stack SRO and object
  --   table sizes.
  --
  --   Finally, the initialize procedure provided by this
  --   module simply calls the
  --   Complete_process_initialization procedure in each
  --   instantiation of Process_Definitions.Process_Instance

  procedure pr1main;
  procedure pr2main;
  procedure pr3main;
  procedure pr4main;

  ----------------------------------------
  -- SPECIFICATIONS FOR USER PROCESS 1 --
  ----------------------------------------

  package UPROC1 is new Process_Definitions.Process_Instance
    (Process_Startup => pr1main,
     SRO_size => 4000,
   service_period =>
            basic_process_management.infinite_period_count);
```

```
-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 2 --
-------------------------------------

package UPROC2 is new Process_Definitions.Process_Instance
  (Process_Startup => pr2main,
   SRO_size => 4000,
   service_period =>
            basic_process_management.infinite_period_count);


-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 3 --
-------------------------------------

package UPROC3 is new Process_Definitions.Process_Instance
  (Process_Startup => pr3main,
   SRO_size => 4000,
   service_period =>
            basic_process_management.infinite_period_count);

-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 4 --
-------------------------------------

package UPROC4 is new Process_Definitions.Process_Instance
  (Process_Startup => pr4main,
   SRO_size => 4000,
   service_period =>
            Basic_Process_Management.Infinite_Period_count);

  ------------------------------
  -- BODIES FOR USER PROCESS 1 --
  ------------------------------

  procedure pr1main is
  begin
    User_Process_1.MAIN;
  end pr1main;

------------------------------
-- BODIES FOR USER PROCESS 2 --
------------------------------

procedure pr2main is
begin
  User_Process_2.main;
end pr2main;
```

```
------------------------------
-- BODIES FOR USER PROCESS 3 --
------------------------------

procedure pr3main is
begin
  User_Process_3.main;
end pr3main;


------------------------------
-- BODIES FOR USER PROCESS 4 --
------------------------------

procedure pr4main is
begin
  User_Process_4.main;
end pr4main;

  ------------------------------------------------
  -- PROCEDURE TO INITIALIZE ALL USER PROCESSES --
  ------------------------------------------------
  procedure Initialize
  is
  begin
    UPROC1.Complete_process_initialization;
--  UPROC2.Complete_process_initialization;
--  UPROC3.Complete_process_initialization;
--  UPROC4.Complete_process_initialization;
  end Initialize;

end User_Processes;
```

INFPSERP2.MBS

```
Pragma environment("ACS:BPM.MLE","ACS:V1USRP.MLI",
    "PR1MAIN.MSE","PR2MAIN.MSE","PR3MAIN.MSE","PR4MAIN.MSE");
with Process_Definitions, iMAX_Definitions,
                                    Basic_Process_management;
with User_Process_1,User_Process_2,User_Process_3,
                                                User_Process_4;
package body User_Processes is

  -- Function:
  --   This module instantiates the user processes and
  --   provides an initialize procedure which completes the
  --   initialization of each user process.
  --
  --   For each user process, there is an instance of
  --   iMAX_Definitions.procedure_val which contains the
  --   process' initial procedure.  To minimize the size of
  --   this package, the initial procedure here simply calls
  --   the actual initial procedure which is found in
  --   another module.  For each process, there is also an
  --   instantiation of Process_Definitions.Process_Instance
  --   which specifies the process id, the procedure_val
  --   just mentioned, and the process' stack SRO and object
  --   table sizes.
  --
  --   Finally, the initialize procedure provided by this
  --   module simply calls the
  --   Complete_process_initialization procedure in each
  --   instantiation of Process_Definitions.Process_Instance

  procedure pr1main;
  procedure pr2main;
  procedure pr3main;
  procedure pr4main;

  ----------------------------------------
  -- SPECIFICATIONS FOR USER PROCESS 1 --
  ----------------------------------------

  package UPROC1 is new Process_Definitions.Process_Instance
    (Process_Startup => pr1main,
     SRO_size => 4000,
   service_period =>
            basic_process_management.infinite_period_count);
```

```
--------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 2 --
--------------------------------------

package UPROC2 is new Process_Definitions.Process_Instance
  (Process_Startup => pr2main,
   SRO_size => 4000,
   service_period =>
            basic_process_management.infinite_period_count);

--------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 3 --
--------------------------------------

package UPROC3 is new Process_Definitions.Process_Instance
  (Process_Startup => pr3main,
   SRO_size => 4000,
   service_period =>
            basic_process_management.infinite_period_count);

--------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 4 --
--------------------------------------

package UPROC4 is new Process_Definitions.Process_Instance
  (Process_Startup => pr4main,
   SRO_size => 4000,
   service_period =>
            Basic_Process_Management.Infinite_Period_count);

  ------------------------------
  -- BODIES FOR USER PROCESS 1 --
  ------------------------------

  procedure pr1main is
  begin
    User_Process_1.MAIN;
  end pr1main;


------------------------------
-- BODIES FOR USER PROCESS 2 --
------------------------------

procedure pr2main is
begin
  User_Process_2.main;
end pr2main;
```

```
------------------------------
-- BODIES FOR USER PROCESS 3 --
------------------------------

procedure pr3main is
begin
  User_Process_3.main;
end pr3main;

------------------------------
-- BODIES FOR USER PROCESS 4 --
------------------------------

procedure pr4main is
begin
  User_Process_4.main;
end pr4main;

  ------------------------------------------------
  -- PROCEDURE TO INITIALIZE ALL USER PROCESSES --
  ------------------------------------------------
  procedure Initialize
  is
  begin
    UPROC1.Complete_process_initialization;
    UPROC2.Complete_process_initialization;
--  UPROC3.Complete_process_initialization;
--  UPROC4.Complete_process_initialization;
  end Initialize;

end User_Processes;
```

INFPSERP3.MBS

```
Pragma environment("ACS:BPM.MLE","ACS:V1USRP.MLI",
   "PR1MAIN.MSE","PR2MAIN.MSE","PR3MAIN.MSE","PR4MAIN.MSE");
with Process_Definitions, iMAX_Definitions,
                                   Basic_Process_management;
with User_Process_1,User_Process_2,User_Process_3,
                                                User_Process_4;
package body User_Processes is

  -- Function:
  --   This module instantiates the user processes and
  --   provides an initialize procedure which completes the
  --   initialization of each user process.
  --
  --   For each user process, there is an instance of
  --   iMAX_Definitions.procedure_val which contains the
  --   process' initial procedure.  To minimize the size of
  --   this package, the initial procedure here simply calls
  --   the actual initial procedure which is found in
  --   another module.  For each process, there is also an
  --   instantiation of Process_Definitions.Process_Instance
  --   which specifies the process id, the procedure_val
  --   just mentioned, and the process' stack SRO and object
  --   table sizes.
  --
  --   Finally, the initialize procedure provided by this
  --   module simply calls the
  --   Complete_process_initialization procedure in each
  --   instantiation of Process_Definitions.Process_Instance

  procedure pr1main;
  procedure pr2main;
  procedure pr3main;
  procedure pr4main;

  ----------------------------------------
  -- SPECIFICATIONS FOR USER PROCESS 1 --
  ----------------------------------------

  package UPROC1 is new Process_Definitions.Process_Instance
    (Process_Startup => pr1main,
     SRO_size => 4000,
   service_period =>
            basic_process_management.infinite_period_count);
```

```
-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 2 --
-------------------------------------

package UPROC2 is new Process_Definitions.Process_Instance
  (Process_Startup => pr2main,
   SRO_size => 4000,
   service_period =>
            basic_process_management.infinite_period_count);

-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 3 --
-------------------------------------

package UPROC3 is new Process_Definitions.Process_Instance
  (Process_Startup => pr3main,
   SRO_size => 4000,
   service_period =>
            basic_process_management.infinite_period_count);


-------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 4 --
-------------------------------------

package UPROC4 is new Process_Definitions.Process_Instance
  (Process_Startup => pr4main,
   SRO_size => 4000,
   service_period =>
            Basic_Process_Management.Infinite_Period_count);

  -------------------------------
  -- BODIES FOR USER PROCESS 1 --
  -------------------------------

  procedure pr1main is
  begin
    User_Process_1.MAIN;
  end pr1main;

-------------------------------
-- BODIES FOR USER PROCESS 2 --
-------------------------------

procedure pr2main is
begin
  User_Process_2.main;
end pr2main;
```

```
------------------------------
-- BODIES FOR USER PROCESS 3 --
------------------------------

procedure pr3main is
begin
  User_Process_3.main;
end pr3main;

------------------------------
-- BODIES FOR USER PROCESS 4 --
------------------------------

procedure pr4main is
begin
  User_Process_4.main;
end pr4main;


  ------------------------------------------------
  -- PROCEDURE TO INITIALIZE ALL USER PROCESSES --
  ------------------------------------------------
  procedure Initialize
  is
  begin
    UPROC1.Complete_process_initialization;
    UPROC2.Complete_process_initialization;
    UPROC3.Complete_process_initialization;
--  UPROC4.Complete_process_initialization;
  end Initialize;

end User_Processes;
```

INFPSERP4.MBS

```
Pragma environment("ACS:BPM.MLE","ACS:V1USRP.MLI",
   "PR1MAIN.MSE","PR2MAIN.MSE","PR3MAIN.MSE","PR4MAIN.MSE");
with Process_Definitions, iMAX_Definitions,
                                   Basic_Process_management;
with User_Process_1,User_Process_2,User_Process_3,
                                              User_Process_4;
package body User_Processes is

  -- Function:
  --   This module instantiates the user processes and
  --   provides an initialize procedure which completes the
  --   initialization of each user process.
  --
  --   For each user process, there is an instance of
  --   iMAX_Definitions.procedure_val which contains the
  --   process' initial procedure.  To minimize the size of
  --   this package, the initial procedure here simply calls
  --   the actual initial procedure which is found in
  --   another module.  For each process, there is also an
  --   instantiation of Process_Definitions.Process_Instance
  --   which specifies the process id, the procedure_val
  --   just mentioned, and the process' stack SRO and object
  --   table sizes.
  --
  --   Finally, the initialize procedure provided by this
  --   module simply calls the
  --   Complete_process_initialization procedure in each
  --   instantiation of Process_Definitions.Process_Instance

  procedure pr1main;
  procedure pr2main;
  procedure pr3main;
  procedure pr4main;

  ----------------------------------------
  -- SPECIFICATIONS FOR USER PROCESS 1 --
  ----------------------------------------

  package UPROC1 is new Process_Definitions.Process_Instance
    (Process_Startup => pr1main,
     SRO_size => 4000,
   service_period =>
           basic_process_management.infinite_period_count);
```

```
--------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 2 --
--------------------------------------

package UPROC2 is new Process_Definitions.Process_Instance
  (Process_Startup => pr2main,
   SRO_size => 4000,
   service_period =>
            basic_process_management.infinite_period_count);


--------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 3 --
--------------------------------------

package UPROC3 is new Process_Definitions.Process_Instance
  (Process_Startup => pr3main,
   SRO_size => 4000,
   service_period =>
            basic_process_management.infinite_period_count);

--------------------------------------
-- SPECIFICATIONS FOR USER PROCESS 4 --
--------------------------------------

package UPROC4 is new Process_Definitions.Process_Instance
  (Process_Startup => pr4main,
   SRO_size => 4000,
   service_period =>
            Basic_Process_Management.Infinite_Period_count);

  ------------------------------
  -- BODIES FOR USER PROCESS 1 --
  ------------------------------

  procedure pr1main is
  begin
    User_Process_1.MAIN;
  end pr1main;

------------------------------
-- BODIES FOR USER PROCESS 2 --
------------------------------

procedure pr2main is
begin
  User_Process_2.main;
end pr2main;
```

```
------------------------------
-- BODIES FOR USER PROCESS 3 --
------------------------------

procedure pr3main is
begin
  User_Process_3.main;
end pr3main;


------------------------------
-- BODIES FOR USER PROCESS 4 --
------------------------------

procedure pr4main is
begin
  User_Process_4.main;
end pr4main;

  ------------------------------------------------
  -- PROCEDURE TO INITIALIZE ALL USER PROCESSES --
  ------------------------------------------------
  procedure Initialize
  is
  begin
    UPROC1.Complete_process_initialization;
    UPROC2.Complete_process_initialization;
    UPROC3.Complete_process_initialization;
    UPROC4.Complete_process_initialization;
  end Initialize;

end User_Processes;
```

INLINK1.LKD

```
; instruction test program

link ACS:minmax.eod
ACS:textio.mlo
intio.mso
schar.mso
scnar.mbo
pr1main.mso
pr2main.mso
pr3main.mso
pr4main.mso
pr1main.mbo
pr2main.mbo
pr3main.mbo
pr4main.mbo
infpserp1.mbo
psors.mbo

free(30 in directory)

output infcnar1.eod

objectmap
```

INFLINK2.LKD

```
; instruction test program

link ACS:minmax.eod
ACS:textio.mlo
intio.mso
schar.mso
schar.mbo
pr1main.mso
pr2main.mso
pr3main.mso
pr4main.mso
pr1main.mbo
pr2main.mbo
pr3main.mbo
pr4main.mbo
infpserp2.mbo
psors.mbo

free(30 in directory)

output infchar2.eod

objectmap
```

INFLINK3.LKD

```
; instruction test program

link ACS:minmax.eod
ACS:textio.mlo
intio.mso
schar.mso
schar.mbo
pr1main.mso
pr2main.mso
pr3main.mso
pr4main.mso
pr1main.mbo
pr2main.mbo
pr3main.mbo
pr4main.mbo
infpserp3.mbo
psors.mbo

free(30 in directory)

output infchar3.eod

objectmap
```

INFLINK3.LKD

```
; instruction test program

link ACS:minmax.eod
ACS:textio.mlo
intio.mso
schar.mso
schar.mbo
pr1main.mso
pr2main.mso
pr3main.mso
pr4main.mso
pr1main.mbo
pr2main.mbo
pr3main.mbo
pr4main.mbo
infpserp4.mbo
psors.mbo

free(30 in directory)

output infchar4.eod

objectmap
```

JCHAR1.COM

```
$ada432new
$ida ACS:intio.mss
$ida scnar.mss
$ida schar.mbs
$ida [rogers.common]pr1main.mss
$ida [rogers.common]pr2main.mss
$ida [rogers.common]pr3main.mss
$ida [rogers.common]pr4main.mss
$purge
$ida pr1main.mbs
$ida pr2main.mbs
$ida pr3main.mbs
$ida pr4main.mbs
$purge
$ida [rogers.common]dfpserp1.mbs
$ida [rogers.common]dfpserp2.mbs
$ida [rogers.common]dfpserp3.mbs
$ida [rogers.common]dfpserp4.mbs
$purge
$ida [rogers.common]infpserp1.mbs
$ida [rogers.common]infpserp2.mbs
$ida [rogers.common]infpserp3.mbs
$ida [rogers.common]infpserp4.mbs
$ida [rogers.common]psors.mbs
$purge
$link432 [rogers.common]dflink1
$link432 [rogers.common]dflink2
$link432 [rogers.common]dflink3
$link432 [rogers.common]dflink4
$link432 [rogers.common]inflink1
$link432 [rogers.common]inflink2
$link432 [rogers.common]inflink3
$link432 [rogers.common]inflink4
$purge
$delete *.**c;*
```

LIST OF REFERENCES

1. Myers, Glenford J., Advances in Computer Architecture, John Wiley, 1982., p. 335.

2. Shoop, Darreld Russel and Holdcroft, Richard T., A Comparative Analysis of Intel's 432 General Data Processor and Control Data's AN/AYK-14(V) Computer System, Master's Thesis, Naval Postgraduate School, Monterey, California, 1982.

3. Applegate, David and Coates, Robert, The Intel 432/670 and ADA Performance Benchmarks, Master's Thesis, Naval Postgraduate School, Monterey, California, 1982.

4. Myers, p. 30.

5. Rattner, Justin, and Cox, George, "Object-Based Computer Architecture", Architecture News, pp.4-11, (October 1980).

6. Naval Postgraduate School Report NPS52-83-001,A View of Object-Oriented Programming, by Bruce J. MacLennan, p. 7, February 1983.
1983)., p. 7.

7. Rattner, p. 6.

8. Organic, Elliott Irving, A Programmer's View of the Intel 432 System, McGraw-Hill, 1983.

9. Myers, p. 4.

10. Ibid., p. 117.

11. Idid., pp. 396 - 414.

12. Kodres, Uno R., "Processing Efficiency of a Class of Multicomputer Systems," International Journal of MINI & Microcomputers, (To be publisned). ACTA Press P.O. Box 2481, Anaheim, CA. 92804

13. Intel Corporation, System 432/600 System Reference Manual, 1981.

14. Shoop, pp. 168 - 171.

15. Intel Corporation, Introduction to the iAPX 432 Architecture, 1981., p. 3-2.

BIBLIGORAPHY

Baer, Jean-Loup, Computer Systems Arcnitecture, Computer Science Press, 1980.

Barnes, J. G. P., Programming in Ada, Addison-Wesley, 1982.

Dennis, Jack B. and Van Horn, Earl C., "Programming Semantics For Multiprogrammed Computations", Communications of ACM, (Marcn 1966), pp 143-155.

Hemenway, Jack, "Object-Oriented Design Manages Software Complexity", EDN, pp. 141-146 (August 19, 1981).

Hemenway, Jack, "Memory Segmentation Aids Object-Oriented Design", EDN, pp. 131-136 (September 30, 1981).

Hemenway, Jack, "Examine Programming Objects From Another Viewpoint", EDN pp.175-278 (November 11, 1981).

Hemenway, Jack, "Understand Program Structure to Fathom 432 Operation", EDN, pp. 147-152, (January 5, 1982).

Intel Corporation, iAPX4 432 General Data Processor Architecture Reference Manual, 1981.

Intel Corporation, iMAX 432 Reference Manual, 1981.

Intel Corporation, iAPX 432 Object Primer, 1981.

Intel Corporation, Introduction to the Intel-432 Cross Development System, 1981.

Intel Corporation, Reference Manual for the Ada Programming Language, 1980.

Intel Corporation, Reference Manual for the Intel 432 Extensions to Ada, 1981.

Intel Corporation, Intel-432 CDS VAX/VMS Host User's Guide, 1981.

MacLennan, Bruce, Principles of Programming Languages: Design, Evaluation and Implementation, Holt, Rinehart and Winston, 1983.

MacLennan, Bruce J., "Values and Objects in Programming Languages", SIGPLAN Notices 17, 12 (December 1982), pp. 70-79.

Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules", Communications of the ACM, December 1972.

Robson, David, "Object-Oriented Software Systems", Byte, August 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Defense Logistic Studies Information Exchange U. S. Army Logistics Management Center Fort Lee, Virginia 23801	1
3.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
5.	Professor Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
6.	Associate Professor Bruce J. MacLennan, Code 52 ml Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
7.	Capt. T. F. Rogers, Jr., USN Box 327 Lumberport, WV 26386	1
8.	Maj. Ioannis A. Karadimitropoulos Hellenic Army Delvinou 16 Papagou Athens, Hellas	1
9.	General Alexander S. Karadimitropoulos Hellenic Army Delvinou 16 Papagou Athens, Hellas	1

10. General Staff (Genicon Epitelion Stratou) 1
G-3 (3 Epitelikon Grafeion)
Hellenic Army
Mesogion Aven
Athens, Hellas

11. Maj. Stavros A. Karadimitropoulos 1
Hellenic Army
KEBOP
Xaidari
Athens, Hellas

12. T. F. Rogers 1
Route 1, Box 352
Lumberport, WV 26386

13. R. J. Summers 1
10592 Spotted Horse Lane
Columbia, MD 21044

14. Maj. James Ervin Vesely, USMC 1
USMC Central Design &
Programming Avtivity
USMC Logistics Base
Albany, GA 31704

15. E. W. Rogers 1
Route 4 Box 285
Clarksburg, WV 25404

16. Cpt. Jonathan Clarke White, USMC 1
Computer Science Department
U. S. Naval Academy
Annapolis, MD 21401

17. Michael Murphy 1
Box 396
Lumberport, WV 26386

18. Dr. Charles R. Arnold 1
Naval Underwater Systems Center
New London, CT 06320

20. B. L. Rogers 1
Route 3 Box 111BA
Martinsburg, WV 25404

21. Daniel Green (Code N20E) 1
Naval Surface Warfare Center
Dahlgren, VA. 22449

22. Capt. V. P. Merz, USN 1
206 Split Oak Place
California, MD 20619

23. Cdr. John Pfeiffer, USN, Code 37 1
Department of Computer Science
Naval Postgraduate School
Monterey, California 93940

24. Cdr J Donegan, USN 1
PMS 400B5
Naval Sea Systems Command
Washington, DC 20362

24. Mike McGowan 1
3585 198 Ave.
Aloha, Oregon 97007

25. Dr. M. J. Gralia 1
Applied Physics Laboratory
Johns Hopkins Road
Laurel, Maryland 20707

26. Dana Small 1
Code 8242
NOSC, San Diego, CA 92152

202310
Thesis
R6858 Rogers
c.1 INTEL 432/670 Ada benchmark performance evaluation in the multi-processor/multiprocess environment.